



Triakis Corporation

Software Design Document

For the Project Entitled:

Shuttle Remote Manipulator System

**A NASA CI03
SARP Initiative 583
IVV-70 Project**



Table of Contents

1	Introduction	3
1.1	System purpose	3
1.2	System scope	3
1.3	Definitions, acronyms, and abbreviations.....	3
1.4	References	4
1.5	SRMS overview	4
2	General system description	5
2.1	System design description	5
2.2	RMA description.....	5
3	Software design description	5
3.1	Initialization.....	7
3.2	Receive SPI Data from Control Panel	15
3.3	Retrieve current data from AFDX devices	17
3.4	Process ‘Manual’ and ‘Angle Seek’ commands.....	24
3.5	Camera & lighting control.....	33
3.6	Strain gauge, device fault, and video image management.....	38
3.7	Update RMA joint displays on Control Panel.....	41
3.8	Process incoming Control Panel data	46
3.9	Header Files.....	49

Table of Figures

Figure 1:	Shuttle RMS Block Diagram	6
-----------	---------------------------------	---



1 Introduction

This specification has been developed to support a research project funded by the NASA Software Assurance Research Program (SARP) during the fiscal year 2003. A system-level, executable specification (ES) based simulation of the Shuttle Remote Manipulator System (SRMS) has been created from the requirements specified in the System Requirements (SARP-I583-001) and Simulator Requirements (SARP-I583-002) Specifications, and will be used as a vehicle for exploring the concepts described in section 2 of Triakis proposal number TC_G020614.

This document describes the software design developed to implement the SRMS as specified in the System Design Document (SARP-I583-101). A simulator will be created and used to evaluate the extent to which the Triakis concept of Executable Specifications (ES') achieves unambiguous communication of system requirements thereby reducing errors induced by interpretation of ambiguous specifications. It will also be used to evaluate the potential that substituting a detailed executable (DE) hardware simulation running actual embedded software, in place of the ES, has for reducing costs and maintaining test consistency through reuse of unmodified system level tests.

Further, new methods of gathering software metrics through use of the simulator will be sought, explored, and evaluated. The virtual system simulator developed for this project will be used to evaluate other potential benefits that its virtual system integration laboratory (VSIL) environment offers in support of general testability, independent validation & verification (IV&V), reliability, and safety.

As our project effort progresses, this specification will be updated to reflect changes to the scope and fidelity of system requirements due to an improved understanding of the extent that our virtual SRMS must be developed to support our research goals.

1.1 System purpose

The system specified herein is intended to represent the SRMS in a general sense only. The software design described in this document will support the development of the RMS Computer DE that will be used as a vehicle to facilitate the research goals stated in Triakis proposal number TC_G020614. The virtual system simulator developed for this project will be used as a vehicle to facilitate the research goals stated in Triakis proposal number TC_G020614. As such, system components and functions of the real-world SRMS that are not required to support our research goals have been omitted.

While the purpose of the actual SRMS is to facilitate the deployment and retrieval of shuttle payloads as well as extra-vehicular activity missions, the derivative SRMS will not incorporate functioning end-effectors required for these purposes. The specified SRMS will demonstrate limited control and movement capability of the RMA along with simulated cameras and video monitors showing the RMA position.

1.2 System scope

The SRMS approximately models a subset of the system characteristics of the existing NASA space shuttle RMS. Adaptations to the functionality of the actual SRMS have been incorporated to the extent required for the stated research purposes and demonstration of the research results.

1.3 Definitions, acronyms, and abbreviations

AFDX	Avionics Full Duplex Switched Ethernet
CCTV	Closed-Circuit Television
CI03	Center Initiative for fiscal year 2003
C/W	Caution/Warning



DE	Detailed Executable
ES	Executable Specification
EVA	Extra Vehicular Activity
IV&V	Independent Verification and Validation
N/A	Not Applicable
NASA	National Aeronautics & Space Administration
OSMA	Office of Safety and Mission Assurance
PDRS	Payload Deployment and Retrieval System
RHC	Rotational Hand Controller
RMA	Remote Manipulator Arm
RMS	Remote Manipulator System
RMSC	RMS Computer
RMSCP	RMS Control Panel
SARP	Software Assurance Research Program
SimRS	Simulator Requirements Specification
SRMS	Shuttle Remote Manipulator System
SyDD	System Design Document
SyRS	System Requirements Specification
THC	Translational Hand Controller
VSIL	Virtual System Integration Laboratory

1.4 References

<http://science.ksc.nasa.gov/shuttle/technology/sts-newsref/sts-deploy> NASA PDRS web page
SARP-I583-001 System Requirements Specification for the Shuttle Remote Manipulator System
SARP-I583-002 Simulator Requirements Specification for the Shuttle Remote Manipulator System
SARP-I583-101 System Design Document for the Shuttle Remote Manipulator System
SARP-I583-102 Software Design Requirements for the Shuttle Remote Manipulator System
SARP-I583-103 ES Implementation Document for the Shuttle Remote Manipulator System
SARP-I583-201 Hardware Design Document for the Shuttle Remote Manipulator System
TC_G020614 Triakis proposal to NASA for the SARP (Solicitation No: NRA SARP 0201), 14 June 2002
RMSComputer.cpp RMS Computer ES code
RMSComputer.h RMS Computer ES code header file
ControlPanel_ICD.h Header file defining control panel button codes, etc.
ControlProcess.cpp Primary RMS Computer target software source code file
ControlProcess.h RMS Computer target software header file

1.5 SRMS overview

Please refer to the NASA [PDRS](#) web page for a more complete description of the real space shuttle SRMS that this system is designed to resemble. The following excerpt is included for quick reference:

The payload deployment and retrieval system (PDRS) includes the electromechanical arm that maneuvers a payload from the payload bay of the space shuttle orbiter to its deployment position and then releases it. It can also grapple a free-flying payload, maneuver it to the payload bay of the orbiter and berth it in the orbiter. This arm is referred to as the remote manipulator system (RMS).

The shuttle RMS is installed in the payload bay of the orbiter for those missions requiring it. Some payloads carried aboard the orbiter for deployment do not require the RMS.

The RMS is capable of deploying or retrieving payloads weighing up to 65,000 pounds. The RMS can also be used to retrieve, repair and deploy satellites; to provide a mobile extension ladder for extravehicular activity crew members for work stations or foot restraints; and to be used as an inspection aid to allow the flight crew members to view the orbiter's or payload's surfaces through a television camera on the RMS.



2 General system description

The system design is described in the System Design Document (SARP-I583-101) for this project. This section gives a high-level overview that is intended to facilitate the understanding of the software design.

2.1 System design description

Power is supplied to the SRMS from the 115vac main shuttle avionics power bus via a circuit breaker on the Power Control Panel. The RMS Computer converts the incoming power to DC voltages suitable to power its own electronics as well as those within the RMS Control Panel. The Power Control Panel supplies power to the shuttle bay cameras and the Remote Manipulator Arm as well. [Figure 1](#) contains a block diagram of the Shuttle Remote Manipulator System.

The RMS Computer communicates with the Remote Manipulator Arm and the cameras in the shuttle bay via Avionics Full Duplex Switched Ethernet (AFDX) serial high-speed databases. In addition to commands and status information, digital compressed video from the CCTV cameras are conveyed over these databases.

The RMS Computer converts the compressed digital camera video signals into RGB format to drive the video inputs of the two video display monitors located on the RMS Control Panel. The RMS Computer communicates with the RMS Control Panel via the Serial Peripheral Interface bus.

2.2 RMA description

The RMA is implemented with 6 degrees of freedom corresponding roughly to the joints of the human arm i.e.: shoulder yaw & pitch joints; elbow pitch joint; and wrist pitch, yaw, & roll joints.

The SRMS design incorporates five CCTV video cameras as specified in the System Requirements Specification. Each of the cameras is equipped with pan, tilt, and zoom capability in addition to featuring a controllable light source. The cameras are located as stated in the SyRS i.e.:

- One on the RMA upper arm boom,
- One on the RMA lower arm boom,
- One at the RMA wrist joint,
- One at the aft wall of the shuttle bay, and
- One at the forward wall of the shuttle bay.

The RMS Control Panel incorporates two video display monitors for displaying CCTV video from any of the five video cameras.

3 Software design description

The software that we developed to control the SRMS within the RMS Computer DE implements the requirements and behavior established in the RMS Computer executable specification. A description of the ES implementation from which this software was developed may be found in the ES Implementation Document (SARP-I583-103). The source file name for the RMS Computer ES code is “RMSComputer.cpp”.

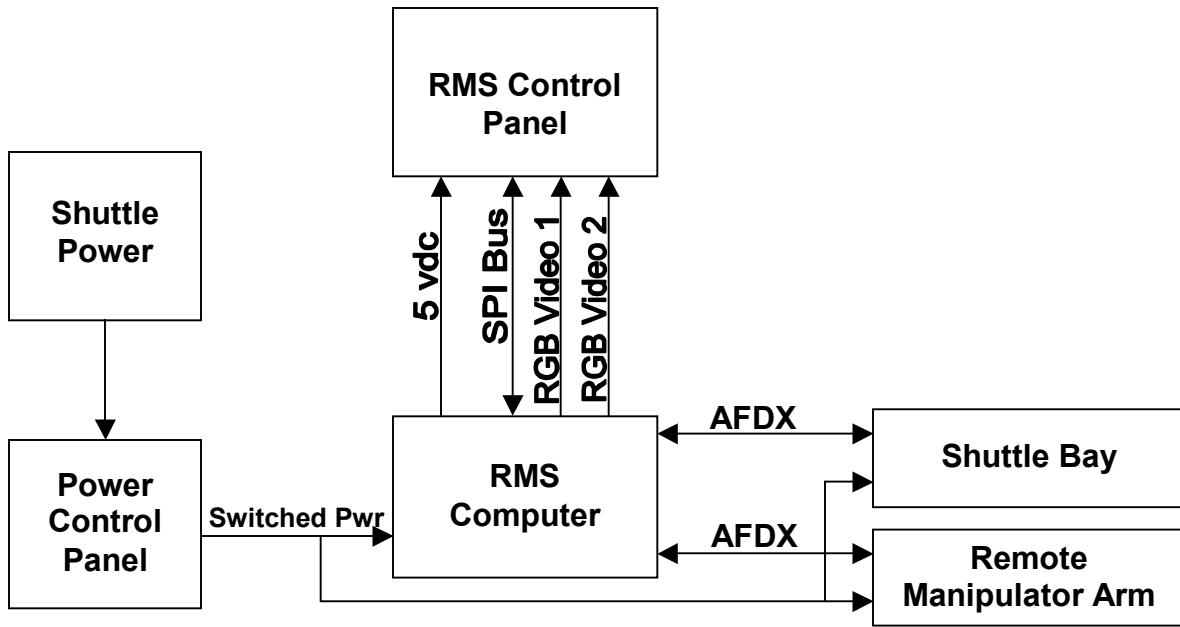


Figure 1: Shuttle RMS Block Diagram

Since the ES is written in C and C++, it is a fairly straightforward process to translate the intent of the ES code into C code to implement the specified behavior in the target hardware environment. The target hardware for which the software is being designed is described in the Hardware Design Document (SARP-I583-101). The source file name for the RMS Computer target code is “ControlProcess.cpp”.

The RMS Computer software executes the following sequence of operations and then repeats steps ‘d’ – ‘i’ on an event-driven periodic basis:

- a) Initialization of variables, constants, pointers, etc.
- b) Receive RMS Control Panel data via SPI receiver if available.
- c) Retrieve current joint angle data from motor controllers.
- d) Process and execute Manual and Angle Seek commands from the RMS Control Panel.
- e) Load video image buffers with fresh data from all video cameras.
- f) Convert & send selected camera images to Control Panel displays.
- g) Update motor angle, seek angle, and rate displays on Control Panel.
- h) Check for, and store incoming Control Panel data via SPI receiver if available.
- i) Loop to step ‘d’.

The following sections show the software source code written to implement the functions listed above and the corresponding ES code from which it was derived. While sequence of operations performed by the target software is roughly the same as that performed by the ES code, there is not a 1:1 correlation between the two.

For the most part, the highlighted comments sprinkled liberally throughout the target SW and ES code describe the actions of program statements to which they relate. Additional descriptions may be included prior to the target SW and ES code listings to aid in understanding the design.



3.1 Initialization

The following initialization file segments are self-descriptive with the comments inserted for clarity. Since the hardware design has implemented the AFDX interface in a pseudo hardware chip form, the target SW includes statements to initialize pointers within the interface chip.

Target SW Initialization

```
void ControlProcess::Init()
{
    // Implement Initialization Requirements

    short joint, cam, axis;
    short ctrlFn, btn, bufPtr;
    short ga, ch;

    // Initialize arrays for storing RMA joint motor data & flags
    for(joint = 0; joint < NumRMAJoints; joint++) {
        jointControlState[joint] = GetJointAngle;
        freshAngleData[joint] = FALSE;
        jointAngleRqstd[joint] = FALSE;
        measAngle[joint] = 0;
        measAngleFract[joint] = 0;
        seekAngle[joint] = 0;
        seekAngleFract[joint] = 0;
        newSeekAngle[joint] = FALSE;
        measVel[joint] = 0.0;
        lastVel[joint] = 0.0;
        motorSpeedCmd[joint] = ZERO;
        rmaJointStatus[joint] = 0;
        activeBuffSize[joint] = 1;
        for(bufPtr=0; bufPtr < ANGLE_HIST_SIZE; bufPtr++)
            angleHist[joint][bufPtr] = 0.0;
    }
    // Initialize arrays for storing camera lamp & motor data & flags
    for(cam = 0; cam < NumCameras; cam++) {
        imgSensorStatus[cam] = 0;
        lampIntensity[cam] = ZERO;
        chgLampInten[cam] = FALSE;
        for(axis = 0; axis < NumCamAxis; axis++) {
            freshCamAngle[cam][axis] = FALSE;
            camAngleRqstd[cam][axis] = FALSE;
            camAngle[cam][axis] = 0;
            camAxisSpeedCmd[cam][axis] = ZERO;
            chgCamAxisSpd[cam][axis] = FALSE;
            camAxisStatus[cam][axis] = 0;
        }
    }
    // Initialize camera control button array
    for(ctrlFn = 0; ctrlFn < NumCtrlFns; ctrlFn++)
        /* 0==Pan==Yaw, 1==Tilt==Pitch, 2==Zoom, 3==Lights */
        for(btn = 0; btn < 2; btn++)
            camCtrlBtns[ctrlFn][btn] = 0;
}
```



Target SW Initialization

```
// Init strain gauge module status
for(ga = 0; ga < NumStrainGauges; ga++)
    strnGaModuleStatus[ga] = 0;
for(ch = 0; ch < NUM_CHANNELS; ch++)
    strnGaData[ga][ch] = 0;

// Initialize array of AFDX motor controller addresses
// (indices are enumerated in the 'ControlPanel_ICD.h' header file)

// RMA joint motor controller addresses
rmaMotorAFDXAddr[ShoulderYaw] = 0x210;
rmaMotorAFDXAddr[ShoulderPitch] = 0x200;
rmaMotorAFDXAddr[ElbowPitch] = 0x300;
rmaMotorAFDXAddr[WristPitch] = 0x400;
rmaMotorAFDXAddr[WristYaw] = 0x410;
rmaMotorAFDXAddr[WristRoll] = 0x420;

// Camera motor controller addresses
camMotorAFDXAddr[LowerArmCam][Pitch] = 0x771;
camMotorAFDXAddr[LowerArmCam][Yaw] = 0x772;
camMotorAFDXAddr[LowerArmCam][Zoom] = 0x773;

camMotorAFDXAddr[UpperArmCam][Pitch] = 0x781;
camMotorAFDXAddr[UpperArmCam][Yaw] = 0x782;
camMotorAFDXAddr[UpperArmCam][Zoom] = 0x783;

camMotorAFDXAddr[WristCam][Pitch] = 0x431;
camMotorAFDXAddr[WristCam][Yaw] = 0x432;
camMotorAFDXAddr[WristCam][Zoom] = 0x433;

camMotorAFDXAddr[FwdBayCam][Pitch] = 0x141;
camMotorAFDXAddr[FwdBayCam][Yaw] = 0x142;
camMotorAFDXAddr[FwdBayCam][Zoom] = 0x143;

camMotorAFDXAddr[AftBayCam][Pitch] = 0x151;
camMotorAFDXAddr[AftBayCam][Yaw] = 0x152;
camMotorAFDXAddr[AftBayCam][Zoom] = 0x153;

// Camera image sensor addresses
imageSensorAFDXAddr[UpperArmCam] = 0x785;
imageSensorAFDXAddr[LowerArmCam] = 0x775;
imageSensorAFDXAddr[WristCam] = 0x435;
imageSensorAFDXAddr[FwdBayCam] = 0x145;
imageSensorAFDXAddr[AftBayCam] = 0x155;

// Camera lamp addresses
camLampAFDXAddr[UpperArmCam] = 0x784;
camLampAFDXAddr[LowerArmCam] = 0x774;
camLampAFDXAddr[WristCam] = 0x434;
camLampAFDXAddr[FwdBayCam] = 0x144;
```



Target SW Initialization

```
camLampAFDXAddr[AftBayCam] = 0x154;  
  
// Strain gauge data module addresses  
// Indices in 'RMSComputer.h'  
strainGaugeAFDXAddr[UpperArm] = 0x786;  
strainGaugeAFDXAddr[LowerArm] = 0x776;  
  
// Set travel limits for camera axis'  
CamPosAngLimit[LowerArmCam][Yaw] = 180;  
CamNegAngLimit[LowerArmCam][Yaw] = 0;  
CamPosAngLimit[LowerArmCam][Pitch] = 180;  
CamNegAngLimit[LowerArmCam][Pitch] = 0;  
CamPosAngLimit[LowerArmCam][Zoom] = 180;  
CamNegAngLimit[LowerArmCam][Zoom] = 30;  
  
CamPosAngLimit[UpperArmCam][Yaw] = 180;  
CamNegAngLimit[UpperArmCam][Yaw] = 0;  
CamPosAngLimit[UpperArmCam][Pitch] = 180;  
CamNegAngLimit[UpperArmCam][Pitch] = 0;  
CamPosAngLimit[UpperArmCam][Zoom] = 180;  
CamNegAngLimit[UpperArmCam][Zoom] = 30;  
  
CamPosAngLimit[WristCam][Yaw] = 180;  
CamNegAngLimit[WristCam][Yaw] = 0;  
CamPosAngLimit[WristCam][Pitch] = 180;  
CamNegAngLimit[WristCam][Pitch] = 0;  
CamPosAngLimit[WristCam][Zoom] = 180;  
CamNegAngLimit[WristCam][Zoom] = 30;  
  
CamPosAngLimit[FwdBayCam][Yaw] = 180;  
CamNegAngLimit[FwdBayCam][Yaw] = 0;  
CamPosAngLimit[FwdBayCam][Pitch] = 180;  
CamNegAngLimit[FwdBayCam][Pitch] = 0;  
CamPosAngLimit[FwdBayCam][Zoom] = 180;  
CamNegAngLimit[FwdBayCam][Zoom] = 30;  
  
CamPosAngLimit[AftBayCam][Yaw] = 180;  
CamNegAngLimit[AftBayCam][Yaw] = 0;  
CamPosAngLimit[AftBayCam][Pitch] = 180;  
CamNegAngLimit[AftBayCam][Pitch] = 0;  
CamPosAngLimit[AftBayCam][Zoom] = 180;  
CamNegAngLimit[AftBayCam][Zoom] = 30;  
  
// Fault mask assignments for control panel lamp display  
masterFaultMask = 0x1;  
faultMask[ShoulderYaw] = 0x2;  
faultMask[ShoulderPitch] = 0x4;  
faultMask[ElbowPitch] = 0x8;  
faultMask[WristPitch] = 0x10;  
faultMask[WristYaw] = 0x20;  
faultMask[WristRoll] = 0x40;
```



Target SW Initialization

```
// Init variables
rtos_init = FALSE;
frame_count = 0;
angleUpdateCnt = 10;
velocityUpdateCnt = 15;
current = 0;
angleHistCnt = 0;
lampRateCnt = 0;

resetFault = FALSE;
cpFaultWord = 0;
lastFaultWd = 0;
errorMsg = 0;
analogChanNum = 0;

// Init control panel switch states
mode = ManualCM;
selectedMotor = ShoulderYaw;
manualCmd = STOP;
modeChange = TRUE;

cmdChange = TRUE;
camCtrlChg = FALSE;
activeCam = AftBayCam;

vidsrc1 = AftBayCam;
vidsrc2 = UpperArmCam;
vidphase = 0; // Video display interlace line number

// Init Pseudo AFDX pointers
afdxbase = (unsigned long *)0x02000000;
unsigned long MotCmdMemBase = 0x0;
unsigned long MotCmdPtrBase = 0x1000;
unsigned long ImgPtrBase = 0x2000;
unsigned long AFDXCmdWd0 = 0x3000;
unsigned long AFDXCmdWd1 = 0x3001;
unsigned long AFDXCmdWd2 = 0x3002;
unsigned long AFDXMsgType = 0x4000;
unsigned long AFDXDestAddr = 0x4001;

// Init joint motor command pointers (pointing to motor command memory)
for(joint = 0; joint < NumRMAJoints; joint++)
    *(afdxbase + 0x1000 + rmaMotorAFDXAddr[joint]) = AFDX_PKT_SIZE * joint;

// Init camera motor command pointers (pointing to motor command memory)
for(cam = 0; cam < NumCameras; cam++)
    for(axis = 0; axis < NumCamAxis; axis++)
        *(afdxbase + 0x1000 + camMotorAFDXAddr[cam][axis]) = AFDX_PKT_SIZE * (cam * NumCamAxis +
axis + NumRMAJoints);

// Init strain gage command pointers (pointing to sg command memory)
for(ga = 0; ga < NumStrainGauges; ga++)
```



Target SW Initialization

```
*(afdxbase + 0x1000 + strainGaugeAFDXAddr[ga]) = AFDX_PKT_SIZE * (ga + NumRMAJoints +  
NumCameras * NumCamAxis);  
  
// Init image sensor command pointers  
for(cam = 0; cam < NumCameras; cam++)  
*(afdxbase + 0x1000 + imageSensorAFDXAddr[cam]) = AFDX_PKT_SIZE *  
(cam + NumRMAJoints + NumCameras * NumCamAxis + NumStrainGauges);  
  
// Init image buffer data pointers  
// (pointing to half megabyte video image buffers)  
for(cam = 0; cam < NumCameras; cam++)  
*(afdxbase + 0x2000 + imageSensorAFDXAddr[cam]) = (512 * 1024) * cam;  
}
```

ES Initialization Code

```
void RMSComputer::Init()  
{  
// 3.1 Hardware Initialization Requirements  
// This function initializes all of the variables and  
// addresses required for RMS Computer and SRMS system  
// initialization.  
  
AFDXSig afdxsigout;  
short joint, cam, axis;  
short ctrlFn, btn, bufPtr;  
short ga, ch;  
  
_cs_1 = TRUE;  
wordcount = 0;  
  
// Initialize arrays for storing joint motor data & flags  
for(joint = 0; joint < NumRMAJoints; joint++) {  
jointControlState[joint] = GetJointAngle;  
freshAngleData[joint] = FALSE;  
jointAngleRqstd[joint] = FALSE;  
measAngle[joint] = 0;  
measAngleFract[joint] = 0;  
seekAngle[joint] = 0;  
seekAngleFract[joint] = 0;  
newSeekAngle[joint] = FALSE;  
measVel[joint] = 0.0;  
lastVel[joint] = 0.0;  
motorSpeedCmd[joint] = ZERO;  
rmaJointStatus[joint] = 0;  
activeBuffSize[joint] = 1;  
for(bufPtr=0; bufPtr < ANGLE_HIST_SIZE; bufPtr++)  
angleHist[joint][bufPtr] = 0.0;  
}  
// Initialize arrays for storing camera lamp & motor data & flags  
for(cam = 0; cam < NumCameras; cam++) {  
imgSensorStatus[cam] = 0;  
lampIntensity[cam] = ZERO;
```



ES Initialization Code

```
chgLampInten[cam] = FALSE;
for(axis = 0; axis < NumCamAxis; axis++) {
    freshCamAngle[cam][axis] = FALSE;
    camAngleRqstd[cam][axis] = FALSE;
    camAngle[cam][axis] = 0;
    camAxisSpeedCmd[cam][axis] = ZERO;
    chgCamAxisSpd[cam][axis] = FALSE;
    camAxisStatus[cam][axis] = 0;
}
}
// Initialize camera control button array
for(ctrlFn = Pan; ctrlFn < NumCtrlFns; ctrlFn++)  
/* 0==Pan==Yaw, 1==Tilt==Pitch, 2==Zoom, 3==Lights */
for(btn = 0; btn < 2; btn++)
    camCtrlBtns[ctrlFn][btn] = 0;

// Init strain gauge module status
for(ga = 0; ga < NumStrainGauges; ga++)
    strnGaModuleStatus[ga] = 0;
for(ch = 0; ch < NUM_CHANNELS; ch++)
    strnGaData[ga][ch] = 0;

// 3.1.1 Aircraft Full Duplex Ethernet (AFDX) Device initialization
// This section initializes the AFDX device address storage arrays
// used for communication with all AFDX devices

// Initialize AFDX motor controller address arrays
// (indices are enumerated in the 'ControlPanel_ICD.h' header file)

// RMA joint motor controller addresses
rmaMotorAFDXAddr[ShoulderYaw] = 0x210;
rmaMotorAFDXAddr[ShoulderPitch] = 0x200;
rmaMotorAFDXAddr[ElbowPitch] = 0x300;
rmaMotorAFDXAddr[WristPitch] = 0x400;
rmaMotorAFDXAddr[WristYaw] = 0x410;
rmaMotorAFDXAddr[WristRoll] = 0x420;

// Camera motor controller addresses
camMotorAFDXAddr[LowerArmCam][Pitch] = 0x771;
camMotorAFDXAddr[LowerArmCam][Yaw] = 0x772;
camMotorAFDXAddr[LowerArmCam][Zoom] = 0x773;

camMotorAFDXAddr[UpperArmCam][Pitch] = 0x781;
camMotorAFDXAddr[UpperArmCam][Yaw] = 0x782;
camMotorAFDXAddr[UpperArmCam][Zoom] = 0x783;

camMotorAFDXAddr[WristCam][Pitch] = 0x431;
camMotorAFDXAddr[WristCam][Yaw] = 0x432;
camMotorAFDXAddr[WristCam][Zoom] = 0x433;

camMotorAFDXAddr[FwdBayCam][Pitch] = 0x141;
camMotorAFDXAddr[FwdBayCam][Yaw] = 0x142;
camMotorAFDXAddr[FwdBayCam][Zoom] = 0x143;
```



ES Initialization Code

```
camMotorAFDXAddr[AftBayCam][Pitch] = 0x151;  
camMotorAFDXAddr[AftBayCam][Yaw] = 0x152;  
camMotorAFDXAddr[AftBayCam][Zoom] = 0x153;
```

// Camera image sensor addresses

```
imageSensorAFDXAddr[UpperArmCam] = 0x785;  
imageSensorAFDXAddr[LowerArmCam] = 0x775;  
imageSensorAFDXAddr[WristCam] = 0x435;  
imageSensorAFDXAddr[FwdBayCam] = 0x145;  
imageSensorAFDXAddr[AftBayCam] = 0x155;
```

// Camera lamp addresses

```
camLampAFDXAddr[UpperArmCam] = 0x784;  
camLampAFDXAddr[LowerArmCam] = 0x774;  
camLampAFDXAddr[WristCam] = 0x434;  
camLampAFDXAddr[FwdBayCam] = 0x144;  
camLampAFDXAddr[AftBayCam] = 0x154;
```

// Strain gauge data module addresses

```
// Indices in 'RMSComputer.h'  
strainGaugeAFDXAddr[UpperArm] = 0x786;  
strainGaugeAFDXAddr[LowerArm] = 0x776;
```

// 3.1.2 Camera axis limits initialization

```
// This section gives the requirements,  
// and initializes the arrays used for storing  
// camera axis travel limits
```

// Set travel limits for camera axis'

```
CamPosAngLimit[LowerArmCam][Yaw] = 180;  
CamNegAngLimit[LowerArmCam][Yaw] = 0;  
CamPosAngLimit[LowerArmCam][Pitch] = 180;  
CamNegAngLimit[LowerArmCam][Pitch] = 0;  
CamPosAngLimit[LowerArmCam][Zoom] = 180;  
CamNegAngLimit[LowerArmCam][Zoom] = 30;
```

```
CamPosAngLimit[UpperArmCam][Yaw] = 180;  
CamNegAngLimit[UpperArmCam][Yaw] = 0;  
CamPosAngLimit[UpperArmCam][Pitch] = 180;  
CamNegAngLimit[UpperArmCam][Pitch] = 0;  
CamPosAngLimit[UpperArmCam][Zoom] = 180;  
CamNegAngLimit[UpperArmCam][Zoom] = 30;
```

```
CamPosAngLimit[WristCam][Yaw] = 180;  
CamNegAngLimit[WristCam][Yaw] = 0;  
CamPosAngLimit[WristCam][Pitch] = 180;  
CamNegAngLimit[WristCam][Pitch] = 0;  
CamPosAngLimit[WristCam][Zoom] = 180;  
CamNegAngLimit[WristCam][Zoom] = 30;
```

```
CamPosAngLimit[FwdBayCam][Yaw] = 180;
```



ES Initialization Code

```
CamNegAngLimit[FwdBayCam][Yaw] = 0;
CamPosAngLimit[FwdBayCam][Pitch] = 180;
CamNegAngLimit[FwdBayCam][Pitch] = 0;
CamPosAngLimit[FwdBayCam][Zoom] = 180;
CamNegAngLimit[FwdBayCam][Zoom] = 30;

CamPosAngLimit[AftBayCam][Yaw] = 180;
CamNegAngLimit[AftBayCam][Yaw] = 0;
CamPosAngLimit[AftBayCam][Pitch] = 180;
CamNegAngLimit[AftBayCam][Pitch] = 0;
CamPosAngLimit[AftBayCam][Zoom] = 180;
CamNegAngLimit[AftBayCam][Zoom] = 30;

// 3.1.3 RMS Control Panel variable & array initialization
// This section initializes the variables & arrays used for
// storing control panel switch & indicator state variables

// Init control panel switch states
mode = ManualCM;
selectedMotor = ShoulderYaw;
manualCmd = STOP;
modeChange = TRUE;

cmdChange = TRUE;
camCtrlChg = FALSE;
activeCam = AftBayCam;

vidsrc1 = AftBayCam;
vidsrc2 = UpperArmCam;

// Mask assignments for control panel fault indicators
masterFaultMask = 0x1;
faultMask[ShoulderYaw] = 0x2;
faultMask[ShoulderPitch] = 0x4;
faultMask[ElbowPitch] = 0x8;
faultMask[WristPitch] = 0x10;
faultMask[WristYaw] = 0x20;
faultMask[WristRoll] = 0x40;

// Initialize video data pointer & flag arrays
for(cam = 0; cam < NumCameras; cam++) {
    camBitMap[cam] = NULL;
    camBitMapFresh[cam] = FALSE;
}

// 3.1.4 Initialization of remaining control
// variables & constants

// Init variables
rtos_init = FALSE;
frame_count = 0;
angleUpdateCnt = 10;
```



ES Initialization Code

```
velocityUpdateCnt = 15;  
current = 0;  
angleHistCnt = 0;  
lampRateCnt = 0;  
  
resetFault = FALSE;  
cpFaultWord = 0;  
lastFaultWd = 0;  
errorMsg = 0;  
analogChanNum = 0;  
}
```

3.2 Receive SPI Data from Control Panel

Note that unlike the ES code, the target software does not have to deal with the SPI bus Chip Select signal (labeled ‘_CS_1’) coming into the RMS Computer from the Control Panel. This is because the hardware design has implemented a 16 word serial input register to accumulate all SPI words and sets an internal ‘dataready’ flag after the Chip Select line has transitioned from low to high signaling the end of data transmission.

While not rigorous, this is a good basic example of how the ES can specify interface protocol requirements without directing implementation. A more thorough specification of the requirements would implement timing delays representative of the data transfer speed, data set up and hold times relative to the chip select line, transmission error detection (e.g. parity, checksum, etc.), and anything else that would need to be considered to specify a robust interface protocol.

Target SW SPI Receive Data

```
void ControlProcess::PeriodicFunc()  
{  
    //short rmaJoint, camNum, axisNum;  
    short joint, cam, axis, ctrlFn, ga;  
    int angleDiff;  
    int deltaAng;  
    int direction;  
    int Move;  
    int manCmd;  
    int i;  
    BOOL change_speed = FALSE;  
    BOOL fault_data = FALSE;  
  
    // If the SPI 'dataready' flag is set, retrieve Control Panel data  
    // from the SPI serial input registers and process command  
    unsigned char dataready = *((unsigned char *) (0x01000000));  
    if(dataready) {  
        for(i = 0; i < MAX_SPI_WORDS; i++)  
            spiChan1Data[i] = *((unsigned short *) (0x01000000 + i));  
        ExecuteCommand(); // Process new input from Control Panel  
    }  
}
```



ES SPI Receive Data Code

```
void RMSComputer::HandleSignal(int id, Sig *data)
{
    AFDXSig afdxsignin;
    AFDXSig afdxsigout;
    SPISig spisig;
    BoolSig bsig;
    short joint, cam, axis, ga;

    switch(id) {

        // 3.3 RMS Control Panel Communication
        // Detailed descriptions of the SPI bus message formats are documented in
        // section 3.2 of the SWDD (SARP-I583-102).

        // 3.3.1 RMS Control Panel Input Requirements
        // All communications to & from the control panel shall use the industry
        // standard Serial Peripheral Interface (SPI) data bus. Data reception
        // shall commence following the high to low transition of the
        // chip select (_CS_1) signal. Incoming SPI data words shall be 16 bits
        // in length and cumulatively stored on each successive clock pulse.
        // SPI bus packet size shall not exceed 16 words. The end of an SPI
        // packet shall be indicated by a low to high transition of the
        // chip select signal.

        // Handle SPI input bus Chip Select Signal
        case _CS_1:
            bsig = *((BoolSig *)data);

            // look for chip select high to low transition to start reading command
            if(_cs_1 && !bsig.tf) {
                wordcount = 0; // Prepare for new Control Panel data packet
                _cs_1 = FALSE;
            }
            // look for chip select low to high transition to execute command
            // _CS_1 going high signals end of control panel data reception
            else if(!_cs_1 && bsig.tf) {
                ExecuteCommand(); // Process new input from Control Panel
                _cs_1 = TRUE;
            }
            break; // End _CS_1

        // Handle Incoming Serial SPI data from Control Panel
        case SerialChan_1_In:

            // Accumulate maximum of 16 command words
            // (MAX_SPI_WORDS defined in 'RMSComputer.h')
            // all SPI words are 16 bits long
            if(wordcount < MAX_SPI_WORDS)
                spiChan1Data[wordcount++] = (short)((SPISig *)data)->data;

            break; // End SerialChan_1_In
    }
}
```



3.3 Retrieve current data from AFDX devices

Initialization is complete when data has been received from all AFDX devices.

Target SW AFDX Device Data Retrieval

```
// Look at incoming AFDX joint motor data

unsigned int cmdmem[AFDX_PKT_SIZE];
unsigned long tcmdptr;

// Evaluate data stored from incoming joint motor controllers
for(joint = 0; joint < NumRMAJoints; joint++) {
    tcmdptr = *(afdxbase + 0x1000 + rmaMotorAFDXAddr[joint]);
    for(i = 0; i < AFDX_PKT_SIZE; i++)
        cmdmem[i] = *(afdxbase + tcmdptr + i); // Retrieve AFDX packet
    switch((int)cmdmem[1]) { // What type of data is it?

        case Status: // Status data
            rmaJointStatus[joint] = cmdmem[2];
            break; //End Status

        case JointAngle: // Joint angle data
            rmaJointStatus[joint] = cmdmem[2];
            measAngle[joint] = cmdmem[3];
            measAngleFract[joint] = cmdmem[4];
            freshAngleData[joint] = TRUE;
            jointAngleRqstd[joint] = FALSE;
            break; //End JointAngle

        case MotorVelocity: // Motor shaft velocity
            // Data not currently used
            break; // End MotorVelocity

        case MotorAngle: // Motor shaft position
            // Data not currently used
            break; // End MotorAngle
    }
    // Clear 'data type' so new data will be recognized as fresh
    *(afdxbase + tcmdptr + 1) = None;
}

// Handle AFDX inputs from camera motor controllers
for(cam = 0; cam < NumCameras; cam++) {
    // Check for video data from camera image sensors

    tcmdptr = *(afdxbase + 0x1000 + imageSensorAFDXAddr[cam]);
    for(i = 0; i < AFDX_PKT_SIZE; i++)
        cmdmem[i] = *(afdxbase + tcmdptr + i); // Retrieve AFDX packet
    switch(cmdmem[1]) { // What type of data is it?
```



Target SW AFDX Device Data Retrieval

```
case Status: // Status data
    imgSensorStatus[cam] = cmdmem[2]; // Store status data
    break; //End Status

case CamImage: // Camera image bitmap
    imgSensorStatus[cam] = cmdmem[2]; // Store status data
    break; //End CamImage
}
// Clear 'data type' so new data will be recognized as fresh
*(afdxbase + tcmdptr + 1) = None;

for(axis = 0; axis < NumCamAxis; axis++) {
    tcmdptr = *(afdxbase + 0x1000 + camMotorAFDXAddr[cam][axis]);
    for(i = 0; i < AFDX_PKT_SIZE; i++)
        cmdmem[i] = *(afdxbase + tcmdptr + i); // Retrieve AFDX packet
    switch((int)cmdmem[1]) { // What type of data is it?

case Status: // Status data
    camAxisStatus[cam][axis] = cmdmem[2]; // Store status data
    break;

case JointAngle: // Joint angle data
    camAxisStatus[cam][axis] = cmdmem[2]; // Store status data
    camAngle[cam][axis] = (short)cmdmem[3]; // Store new angle data
    freshCamAngle[cam][axis] = TRUE; // Set fresh angle data flag
    camAngleRqstd[cam][axis] = FALSE;
    break;

case MotorVelocity: // Motor shaft velocity
    // Data not currently used
    break; // End MotorVelocity

case MotorAngle: // Motor shaft position
    // Data not currently used
    break; // End MotorAngle
}
// Clear 'data type' so new data will be recognized as fresh
*(afdxbase + tcmdptr + 1) = None;
}

// Handle AFDX inputs from strain gauge modules
for(ga = 0; ga < NumStrainGauges; ga++) {
    tcmdptr = *(afdxbase + 0x1000 + strainGaugeAFDXAddr[ga]);
    for(i = 0; i < AFDX_PKT_SIZE; i++)
        cmdmem[i] = *(afdxbase + tcmdptr + i); // Retrieve AFDX packet
    switch((int)cmdmem[1]) { // What type of data is it?

case Status: // Status data
    strnGaModuleStatus[ga] = cmdmem[2]; // Store status data
    break; //End Status

case Resistance: // Resistance data
```



Target SW AFDX Device Data Retrieval

```
strnGaModuleStatus[ga] = cmdmem[2]; // Store status data
analogChanNum = cmdmem[3];
strnGaData[ga][analogChanNum] = cmdmem[4];
break; //End Resistance

case CmdError:
strnGaModuleStatus[ga] = cmdmem[2]; // Store status data
analogChanNum = cmdmem[3]; // Analog channel number
errorMsg = cmdmem[4]; // Error code
break; // End CmdError

case SelfTest: // Self test status data
strnGaModuleStatus[ga] = cmdmem[2];
break; //End Status
}
// Clear 'data type' so new data will be recognized as fresh
*(afdxbase + tcmdptr + 1) = None;
}
// Initialization Tasks After RTOS Start

if(!rtos_init) {
rtos_init = TRUE;

if(frame_count == 0) { // Give other AFDX parts time to initialize
// Query status of all AFDX devices
*(afdxbase + 0x3000) = 0x0; // Source address = RMS Computer
*(afdxbase + 0x4000) = QueryStatusAll; // Command type
*(afdxbase + 0x4001) = 0x0; // Dest Addx & Issue command
}

// Get Initial Motor Data
// When all motor data is fresh, initialization is complete
for(joint = 0; joint < NumRMAJoints; joint++)
if(!freshAngleData[joint]) {
rtos_init = FALSE;
break;
}

// And the camera motors too...
for(cam = 0; cam < NumCameras; cam++)
for(axis = 0; axis < NumCamAxis; axis++)
if(!freshCamAngle[cam][axis]) {
rtos_init = FALSE;
break;
}
}

if(frame_count >= angleUpdateCnt) {
angleUpdateCnt += 10; // 10 frames = 1 second
// Update joint & axis data periodically

// Request RMA joint data from motor controllers
```



Target SW AFDX Device Data Retrieval

```
for(joint = 0; joint < NumRMAJoints; joint++) {  
  
    freshAngleData[joint] = FALSE;  
    jointAngleRqstd[joint] = FALSE;  
}  
  
// Request camera axis data from motor controllers  
for(cam = 0; cam < NumCameras; cam++)  
    for(axis = 0; axis < NumCamAxis; axis++) {  
  
        freshCamAngle[cam][axis] = FALSE;  
        camAngleRqstd[cam][axis] = FALSE;  
    }  
}
```

ES AFDX Device Data Retrieval

// 3.4.1 Data Received from AFDX devices

```
case Databus_A_I: // Handle all incoming AFDX data  
    afdxsigin = *((AFDXSig *)data);  
  
// 3.4.1.1 AFDX RMA joint motor controllers  
  
// Handle AFDX inputs from RMA motor controllers  
for(joint = 0; joint < NumRMAJoints; joint++) {  
    if(afdxsigin.data[0] == rmaMotorAFDXAddr[joint]) { // Address match?  
        switch(afdxsigin.data[1]) { // What type of data is it?  
  
            case Status: // Status data  
                rmaJointStatus[joint] = afdxsigin.data[2];  
                break; //End Status  
  
            case JointAngle: // Joint angle data  
                rmaJointStatus[joint] = afdxsigin.data[2];  
                measAngle[joint] = (int)afdxsigin.data[3];  
                measAngleFract[joint] = (int)afdxsigin.data[4];  
                freshAngleData[joint] = TRUE;  
                jointAngleRqstd[joint] = FALSE;  
                break; //End JointAngle  
  
            case MotorVelocity: // Motor shaft velocity  
                // Data not currently used  
                break; // End MotorVelocity  
  
            case MotorAngle: // Motor shaft position  
                // Data not currently used  
                break; // End MotorAngle  
        }  
    }  
}
```



ES AFDX Device Data Retrieval

// 3.4.1.2 AFDX camera image sensors

```
// Handle AFDX inputs from camera motor controllers & image sensors
for(cam = 0; cam < NumCameras; cam++) {

    // Check for video data from camera image sensors
    if(afdxsignin.data[0] == imageSensorAFDXAddr[cam]) { // Address match?
        switch(afdxsignin.data[1]) { // What type of data is it?

            case Status: // Status data
                imgSensorStatus[cam] = afdxsignin.data[2]; // Store status data
                break; //End Status

            case CamImage: // Camera image bitmap
                camBitMap[cam] = (unsigned char *) (afdxsignin.buffPtr);
                imgSensorStatus[cam] = afdxsignin.data[2]; // Store status data
                camBitMapFresh[cam] = TRUE;
                break; //End CamImage
        }
    }
}
```

// 3.4.1.3 AFDX camera axis motor controllers

```
for(axis = 0; axis < NumCamAxis; axis++)
    if(afdxsignin.data[0] == camMotorAFDXAddr[cam][axis]) {
        switch(afdxsignin.data[1]) { // What type of data is it?

            case Status: // Status data
                camAxisStatus[cam][axis] = afdxsignin.data[2]; // Store status data
                break; //End Status

            case JointAngle: // Joint angle data
                camAxisStatus[cam][axis] = afdxsignin.data[2]; // Store status data
                camAngle[cam][axis] = (int)afdxsignin.data[3]; // Store new angle data
                freshCamAngle[cam][axis] = TRUE; // Set fresh angle data flag
                camAngleRqstd[cam][axis] = FALSE;
                break; //End JointAngle

            case MotorVelocity: // Motor shaft velocity
                // Data not currently used
                break; // End MotorVelocity

            case MotorAngle: // Motor shaft position
                // Data not currently used
                break; // End MotorAngle
        }
    }
}
```

// 3.4.1.4 AFDX strain gauge data acquisition modules

```
// Check for data from strain gauge modules
for(ga = 0; ga < NumStrainGauges; ga++)
    if(afdxsignin.data[0] == strainGaugeAFDXAddr[ga]) { // Address match?
```



ES AFDX Device Data Retrieval

```
switch(afdxsigin.data[1]) { // What type of data is it?

    case Status: // Status data
        strnGaModuleStatus[ga] = afdxsigin.data[2];
        break; //End Status

    case Resistance: // Resistance data
        strnGaModuleStatus[ga] = afdxsigin.data[2];
        analogChanNum = afdxsigin.data[3];
        strnGaData[ga][analogChanNum] = afdxsigin.data[4];
        break; //End Resistance

    case CmdError:
        strnGaModuleStatus[ga] = afdxsigin.data[2];
        analogChanNum = afdxsigin.data[3]; // Analog channel number
        errorMsg = afdxsigin.data[4]; // Error code
        break; // End CmdError

    case SelfTest: // Self test status data
        strnGaModuleStatus[ga] = afdxsigin.data[2];
        break; //End Status

    }

}

break; // End Databus_A_I
}
// SendSignal(id, data);
}

// 3.4.1 Data Sent to AFDX devices
// Throughout the following function data is sent to the various AFDX devices
// to implement the system functionality as required. Detailed descriptions
// of the AFDX message formats are documented in section 3.2 of the SWDD
// (SARP-I583-102).

void RMSComputer::HandleEvent()
{
    AFDXSig afdxsigout;
    RGBSig rgbsig;
    SPISig spisig;
    BoolSig bsig;
    short joint, cam, axis, ctrlFn, ga;
    int angleDiff;
    int deltaAng;
    int direction;
    int Move;
    int manCmd;

    // Begin Periodic Real Time Processing

    // Initialization Tasks After RTOS Start
```



ES AFDX Device Data Retrieval

```
if(!rtos_init) {
    rtos_init = TRUE;

    if(frame_count == 0) {
        // Send QueryStatusAll AFDX message to self (address = 0)
        // AFDXsig structure defined in file 'SigAFDX.h'
        AFDXSig afdxsig;
        afdxsig.msgType = AFDXSig::QueryStatusAll;
        SendSignal(Databus_A_O, &afdxsig);
    }

    // Init display bitmaps
    if(camBitMapFresh[vidsrc1]) {
        rgbsig.bitmapptr = camBitMap[vidsrc1];
        SendSignal(RGB_1, &rgbsig);
    }
    else
        rtos_init = FALSE;

    if(camBitMapFresh[vidsrc2]) {
        rgbsig.bitmapptr = camBitMap[vidsrc2];
        SendSignal(RGB_2, &rgbsig);
    }
    else
        rtos_init = FALSE;

    // Check initial motor data
    // When all motor data is fresh, initialization is complete
    for(joint = 0; joint < NumRMAJoints; joint++)
        if(!freshAngleData[joint]) {
            rtos_init = FALSE;
            break;
        }

    // And the camera motors too...
    for(cam = 0; cam < NumCameras; cam++)
        for(axis = 0; axis < NumCamAxis; axis++)
            if(!freshCamAngle[cam][axis]) {
                rtos_init = FALSE;
                break;
            }
    }

    if(frame_count >= angleUpdateCnt) {
        angleUpdateCnt += 10; // 10 frames = 1 second
        // Update joint & axis data periodically

        // Request RMA joint data from motor controllers
        for(joint = 0; joint < NumRMAJoints; joint++) {

            freshAngleData[joint] = FALSE;
            jointAngleRqstd[joint] = FALSE;
        }
    }
}
```

**ES AFDX Device Data Retrieval**

```

}

// Request camera axis data from motor controllers
for(cam = 0; cam < NumCameras; cam++) {
    for(axis = 0; axis < NumCamAxis; axis++) {

        freshCamAngle[cam][axis] = FALSE;
        camAngleRqstd[cam][axis] = FALSE;
    }
}

```

3.4 Process ‘Manual’ and ‘Angle Seek’ commands

The joint motors are programmed to move at a fixed speed of 600 RPM. Since the motors drive the joints through a 500:1 reduction gearbox, the actual joint speed is 1.2 RPM or 7.2 degrees per second. There is no ramp-up or ramp-down period used in our simplified RMA joint movement algorithms and while there are no friction brakes in our joint designs, the motors automatically enter into an active angle-hold mode (a feature of the motor controller itself) when stopped.

The combined ‘Manual’ and ‘Angle Seek’ state machine is summarized below:

- 1) **GetJointAngle:** Enter upon initialization.
Enter from ChangeSpeed state.
Enter from Seeking state if no motor speed change commanded.
 If moving, query motor for joint position update.
Exit to Seeking state if (mode == AngleSeek) AND (*freshAngleData* OR *newSeekAngle*).
Exit to ProcessCmd state if mode == Manual.
- 2) **ProcessCmd:** Enter from GetJointAngle state if Mode == Manual.
 Upon new command from control panel for selected motor:
 If manualCommand == INC, motorSpeedCmd = MaxSpeed_CW;
 If manualCommand == DEC, motorSpeedCmd = MaxSpeed_CCW;
 If manualCommand == STOP, motorSpeedCmd = ZERO;
 Send new motorSpeedCmd if different from previous motorSpeedCmd;
 Stop non-selected motors in motion;
Exit to GetJointAngle state if no change in motorSpeedCmd.
Exit to ChangeSpeed state if change in motorSpeedCmd.
- 3) **Seeking:** Enter from GetJointAngle state if (*freshAngleData* OR *newSeekAngle*).
 Compute shortest distance to seek angle;
 Compute optimal direction (i.e. shortest seek path);
 If shortest distance to seek angle is <= 2 degrees:
 Stop motor if moving;
 Else if motor stopped OR moving in opposite direction:
 Command motor to move in optimal direction;
 Exit to ChangeSpeed state if new motor velocity has been set.
- 4) **ChangeSpeed:** Enter from Seeking state if new motor velocity has been set.
Enter from ProcessCmd state if new motor velocity has been set.
 Send new motor speed command to motor via AFDX ExecuteCommand.
Exit to GetJointAngle state.

Target SW ‘Manual’ & ‘Angle Seek’ Command Processing

```
// Process motor control modes & commands from Control Panel
```



Target SW ‘Manual’ & ‘Angle Seek’ Command Processing

```
if(modeChange) { // Stop all motors when changing command modes
    modeChange = FALSE;

    for(joint = 0; joint < NumRMAJoints; joint++) {
        if(motorSpeedCmd[joint] != ZERO) { // If not already at rest,
            motorSpeedCmd[joint] = ZERO; // Stop motor
            jointControlState[joint] = ChangeSpeed; // Next state
        }
        else {
            jointControlState[joint] = GetJointAngle; // Next state
        }
        seekAngle[joint] = measAngle[joint]; // Reset seekAngle
        seekAngleFract[joint] = measAngleFract[joint];
    }
}

// Begin RMA motor control state machine

for(joint = 0; joint < NumRMAJoints; joint++) {

    switch(jointControlState[joint]) {

        case GetJointAngle:
            if(!freshAngleData[joint])
                // Request fresh data only as needed

                *(afdxbase + 0x3000) = 0x0; // source address = RMS Computer
                *(afdxbase + 0x3001) = JointAngle; // Data type = joint angle
                *(afdxbase + 0x4000) = QueryResponse; // Command type
                *(afdxbase + 0x4001) = rmaMotorAFDXAddr[joint]; // Issue command

                jointAngleRqstd[joint] = TRUE;
            }
        switch(mode) {
            case AngleSeekCM:
                jointControlState[joint] = Seeking; // Next state

                if(cmdChange && (manualCmd == STOP)) {
                    if(joint == selectedMotor) {
                        seekAngle[joint] = measAngle[joint]; // Reset seekAngle
                        cmdChange = FALSE;
                    }
                }
                else if(cmdChange && (manualCmd != STOP))
                    cmdChange = FALSE;

                break;

            case ManualCM:
                jointControlState[joint] = ProcessCmd; // Next state
                break;
        }
    }
}
```



Target SW ‘Manual’ & ‘Angle Seek’ Command Processing

```
}

break; // End of GetJointAngle state

case ProcessCmd: // Process Manual mode commands
    jointControlState[joint] = GetJointAngle; // Next state
    if(joint == selectedMotor) && cmdChange) {

        // Process new command for selected motor
        cmdChange = FALSE;
        switch(manualCmd) {

            case STOP:
                if(motorSpeedCmd[joint] != ZERO) {
                    motorSpeedCmd[joint] = ZERO;
                    jointControlState[joint] = ChangeSpeed; // Next state
                }
                break;

            case INC:
                if(motorSpeedCmd[joint] != MAX_SPEED_CW) {
                    motorSpeedCmd[joint] = MAX_SPEED_CW;
                    jointControlState[joint] = ChangeSpeed; // Next state
                }
                break;

            case DEC:
                if(motorSpeedCmd[joint] != MAX_SPEED_CCW) {
                    motorSpeedCmd[joint] = MAX_SPEED_CCW;
                    jointControlState[joint] = ChangeSpeed; // Next state
                }
                break;
        }
    }
    else // Update angle data for moving motors
        if(joint == selectedMotor) && (motorSpeedCmd[joint] != ZERO))
            freshAngleData[joint] = FALSE; // Reset flag to request fresh data

    else // Motors not selected should be at rest
        if(joint != selectedMotor) && (motorSpeedCmd[joint] != ZERO)) {
            motorSpeedCmd[joint] = ZERO; // Stop motors not selected
            jointControlState[joint] = ChangeSpeed; // Next state
        }
    break; // End of ProcessCmd state

case Seeking: // Angle-Seek algorithm
    jointControlState[joint] = GetJointAngle; // Next state
    // Find shortest distance to seek angle
    angleDiff = seekAngle[joint] - measAngle[joint];

    if(angleDiff < 0) {
        if(angleDiff < -180) {
```



Target SW ‘Manual’ & ‘Angle Seek’ Command Processing

```
direction = CW; // Shortest path is CW
deltaAng = 360 + angleDiff; // Shortest distance
}
else { // 0 > angleDiff >= -180
    direction = CCW; // Shortest path is CCW
    deltaAng = - angleDiff; // Shortest distance
}
}
else if (angleDiff >= 0) {
if (angleDiff > 180) {
    direction = CCW; // Shortest path is CCW
    deltaAng = 360 - angleDiff; // Shortest distance
}
else { // 0 < angleDiff <= 180
    direction = CW; // Shortest path is CW
    deltaAng = angleDiff; // Shortest distance
}
}

// Before starting seek, check to see if motor is within 2 degrees
// of its destination angle. Stop (or don't start) motor if angle
// is within 2 degrees of destination

if((deltaAng <= 2) && ((motorSpeedCmd[joint] != ZERO) ||
    ((newSeekAngle[joint] == TRUE) && (motorSpeedCmd[joint] == ZERO)))) {
    // Stop the joint if it has arrived or
    // if joint is already stopped at newly requested seek angle
    motorSpeedCmd[joint] = ZERO;
    jointControlState[joint] = ChangeSpeed; // Next state
    newSeekAngle[joint] = FALSE;
}
else if((deltaAng <= 2) && (motorSpeedCmd[joint] == ZERO))
    // Joint is already stopped
    jointControlState[joint] = GetJointAngle; // Next state

else { // Start (or keep) seeking
switch(direction) { // Set motor speed & direction

case CW:
    if(motorSpeedCmd[joint] != MAX_SPEED_CW) {
        motorSpeedCmd[joint] = MAX_SPEED_CW;
        jointControlState[joint] = ChangeSpeed; // Next state
    }
    break;

case CCW:
    if(motorSpeedCmd[joint] != MAX_SPEED_CCW) {
        motorSpeedCmd[joint] = MAX_SPEED_CCW;
        jointControlState[joint] = ChangeSpeed; // Next state
    }
    break;
}
}
```



Target SW ‘Manual’ & ‘Angle Seek’ Command Processing

```
freshAngleData[joint] = FALSE; // Reset flag to request fresh data
newSeekAngle[joint] = FALSE;
}
break; // End of Seeking state

case ChangeSpeed: // Change motor speed

    // Build & send command to motor
    *(afdxbase + 0x3000) = 0x0; // source address = RMS Computer
    *(afdxbase + 0x3001) = MotorVelocity; // Data type = motor velocity
    *(afdxbase + 0x3002) = motorSpeedCmd[joint]; // Motor velocity
    *(afdxbase + 0x4000) = ExecuteCmd; // Command type
    *(afdxbase + 0x4001) = rmaMotorAFDXAddr[joint]; // Issue command

    jointControlState[joint] = GetJointAngle; // Next state
    freshAngleData[joint] = FALSE; // Reset flag to request fresh data

    // Determine appropriate command button to light

switch(motorSpeedCmd[joint]) {
case ZERO:
    manCmd = STOP;
    break; // End ZERO

case MAX_SPEED_CCW:
    manCmd = DEC;
    break; // End MAX_SPEED_CCW

case MAX_SPEED_CW:
    manCmd = INC;
    break; // End MAX_SPEED_CW
}

// Send manual cmd button light data to control panel

iio->SetOutData(0, 0, (unsigned char)ManualCmdData);
iio->SetOutData(0, 1, 0); // Upper byte of ManualCmdData always 0
iio->SetOutData(0, 2, (unsigned char)joint);
iio->SetOutData(0, 3, 0); // Upper byte of joint always 0
iio->SetOutData(0, 4, (unsigned char)manCmd);
iio->SetOutData(0, 5, 0); // Upper byte of manCmd always 0

change_speed = TRUE; // only one spi sequence allowed per frame

break; // End of ChangeSpeed state
}
```

ES ‘Manual’ & ‘Angle Seek’ Command Processing Code

```
// 3.4 RMA Control Functions
// The SRMS shall support two manual operational control modes - single-joint control and angle-seek control.
// All joints shall immediately come to rest when the control mode is changed.
```



ES ‘Manual’ & ‘Angle Seek’ Command Processing Code

```
// Process control mode changes from Control Panel

if(modeChange) { // Stop all motors when changing command modes
    modeChange = FALSE;

    for(joint = 0; joint < NumRMAJoints; joint++) {
        if(motorSpeedCmd[joint] != ZERO) { // If not already at rest,
            motorSpeedCmd[joint] = ZERO; // Stop motor
            jointControlState[joint] = ChangeSpeed; // Next state
        }
        else {
            jointControlState[joint] = GetJointAngle; // Next state
        }
        seekAngle[joint] = measAngle[joint]; // Reset seekAngle
        seekAngleFract[joint] = measAngleFract[joint];
    }
}

// 3.4.1 Manual Operation
// Single-joint manual control shall allow the operator to move the RMA one joint at a time.
// The operator shall be able to select any one of the RMA joints to increment/decrement the joint angle, and stop it on command.

// 3.4.2 Angle Seek Operation
// Angle-seek mode shall provide for computer-assisted manual RMA control.
// The computer control shall be optimized so that the joint travels in the direction of the shortest distance to the destination angle.
// The operator shall be able to stop individual joints from moving at any time.

// Both Manual and Angle-Seek command modes are implemented with this state machine
// Begin RMA motor control state machine
// Process motor control modes & commands from Control Panel

for(joint = 0; joint < NumRMAJoints; joint++) {

    switch(jointControlState[joint]) {

        case GetJointAngle:
            if(!freshAngleData[joint]/* && !jointAngleRqstd[joint]*/) {
                // Request fresh data only as needed

                afdxsigout.msgType = AFDXSig:QueryResponse; // Command type
                afdxsigout.address = rmaMotorAFDXAddr[joint]; // Set motor Addx
                afdxsigout.data[1] = JointAngle; // Data type = joint angle
                SendSignal(Databus_A_O, &afdxsigout); // Issue command

                jointAngleRqstd[joint] = TRUE;
            }
        switch(mode) {
            case AngleSeekCM:
                jointControlState[joint] = Seeking; // Next state
```



ES ‘Manual’ & ‘Angle Seek’ Command Processing Code

```
if(cmdChange && (manualCmd == STOP)) {
    if(joint == selectedMotor) {
        seekAngle[joint] = measAngle[joint]; // Reset seekAngle
        cmdChange = FALSE;
    }
}
else if(cmdChange && (manualCmd != STOP))
    cmdChange = FALSE;

break;

case ManualCM:
    jointControlState[joint] = ProcessCmd; // Next state
    break;
}

break; // End of GetJointAngle state

case ProcessCmd: // Process Manual mode commands
    jointControlState[joint] = GetJointAngle; // Next state
    if(joint == selectedMotor) && cmdChange {

        // Process new command for selected motor
        cmdChange = FALSE;
        switch(manualCmd) {

            case STOP:
                if(motorSpeedCmd[joint] != ZERO) {
                    motorSpeedCmd[joint] = ZERO;
                    jointControlState[joint] = ChangeSpeed; // Next state
                }
                break;

            case INC:
                if(motorSpeedCmd[joint] != MAX_SPEED_CW) {
                    motorSpeedCmd[joint] = MAX_SPEED_CW;
                    jointControlState[joint] = ChangeSpeed; // Next state
                }
                break;

            case DEC:
                if(motorSpeedCmd[joint] != MAX_SPEED_CCW) {
                    motorSpeedCmd[joint] = MAX_SPEED_CCW;
                    jointControlState[joint] = ChangeSpeed; // Next state
                }
                break;
        }
    }

else // Update angle data for moving motors
    if(joint == selectedMotor) && (motorSpeedCmd[joint] != ZERO))
        freshAngleData[joint] = FALSE; // Reset flag to request fresh data

else // Motors not selected should be at rest
    if(joint != selectedMotor) && (motorSpeedCmd[joint] != ZERO)) {
```



ES ‘Manual’ & ‘Angle Seek’ Command Processing Code

```
motorSpeedCmd[joint] = ZERO; // Stop motors not selected
jointControlState[joint] = ChangeSpeed; // Next state
}
break; // End of ProcessCmd state

case Seeking: // Angle-Seek algorithm
jointControlState[joint] = GetJointAngle; // Next state
// Find shortest distance to seek angle
angleDiff = seekAngle[joint] - measAngle[joint];

if(angleDiff < 0) {
    if(angleDiff < -180) {
        direction = CW; // Shortest path is CW
        deltaAng = 360 + angleDiff; // Shortest distance
    }
    else { // 0 > angleDiff >= -180
        direction = CCW; // Shortest path is CCW
        deltaAng = - angleDiff; // Shortest distance
    }
}
else if(angleDiff >= 0) {
    if(angleDiff > 180) {
        direction = CCW; // Shortest path is CCW
        deltaAng = 360 - angleDiff; // Shortest distance
    }
    else { // 0 < angleDiff <= 180
        direction = CW; // Shortest path is CW
        deltaAng = angleDiff; // Shortest distance
    }
}

// Before starting seek, check to see if motor is within 2 degrees
// of its destination angle. Stop (or don't start) motor if angle
// is within 2 degrees of destination

if((deltaAng <= 2) && ((motorSpeedCmd[joint] != ZERO) ||
    ((newSeekAngle[joint] == TRUE) && (motorSpeedCmd[joint] == ZERO)))) {
    // Stop the joint if it has arrived or
    // if joint is already stopped at newly requested seek angle
    motorSpeedCmd[joint] = ZERO;
    jointControlState[joint] = ChangeSpeed; // Next state
    newSeekAngle[joint] = FALSE;
}
else if((deltaAng <= 2) && (motorSpeedCmd[joint] == ZERO))
    // Joint is already stopped
    jointControlState[joint] = GetJointAngle; // Next state

else { // Start (or keep) seeking
    switch(direction) { // Set motor speed & direction

        case CW:
            if(motorSpeedCmd[joint] != MAX_SPEED_CW) {
                motorSpeedCmd[joint] = MAX_SPEED_CW;
                jointControlState[joint] = ChangeSpeed; // Next state
            }
    }
}
```



ES ‘Manual’ & ‘Angle Seek’ Command Processing Code

```
}

break;

case CCW:
if(motorSpeedCmd[joint] != MAX_SPEED_CCW) {
    motorSpeedCmd[joint] = MAX_SPEED_CCW;
    jointControlState[joint] = ChangeSpeed; // Next state
}
break;
}
freshAngleData[joint] = FALSE; // Reset flag to request fresh data
newSeekAngle[joint] = FALSE;
}
break; // End of Seeking state

case ChangeSpeed: // Change motor speed

// Build & send command to motor
afdxsigout.msgType = AFDXSig::ExecuteCmd;
afdxsigout.address = rmaMotorAFDXAddr[joint];
afdxsigout.data[1] = MotorVelocity; // Data type = motor velocity
afdxsigout.data[2] = motorSpeedCmd[joint]; // Motor velocity
SendSignal(Databus_A_O, &afdxsigout); // Issue command

// Determine appropriate command button to light

switch(motorSpeedCmd[joint]) {
case ZERO:
    manCmd = STOP;
    break; // End ZERO

case MAX_SPEED_CCW:
    manCmd = DEC;
    break; // End MAX_SPEED_CCW

case MAX_SPEED_CW:
    manCmd = INC;
    break; // End MAX_SPEED_CW
}

// 3.3.2.1 Manual Command button group back-light control
// The RMS Computer shall light the appropriate control panel manual
// command lamp when moving the associated RMA joint.

bsig.tf = FALSE;
SendSignal(_CS_DATA_1, &bsig);
spisig.data = ManualCmdData;
SendSignal(SerialChan_Data_1_Out, &spisig);
spisig.data = joint;
SendSignal(SerialChan_Data_1_Out, &spisig);
spisig.data = manCmd; // Target manual command button
SendSignal(SerialChan_Data_1_Out, &spisig);
```



ES ‘Manual’ & ‘Angle Seek’ Command Processing Code

```
bsig.tf = TRUE;  
SendSignal(_CS_DATA_1, &bsig);  
  
jointControlState[joint] = GetJointAngle; // Next state  
freshAngleData[joint] = FALSE; // Reset flag to request fresh data  
  
break; // End of ChangeSpeed state  
  
}  
} // End RMA joint command processing state machine
```

3.5 Camera & lighting control

This code segment processes camera and lighting control command inputs from the RMS Control Panel and controls the camera and camera lights in response. The lighting function has not been implemented in the lamp parts but the commands are in place to support it when the function is added.

Target SW

```
// Process camera control commands  
  
if(camCtrlChg) { // Change camera movement only when directed  
    camCtrlChg = FALSE;  
  
    // Process Pan, Tilt, & Zoom control buttons  
    // 0==Pan==Yaw, 1==Tilt==Pitch, 2==Zoom  
    for(ctrlFn = Pan; ctrlFn <= Zoom; ctrlFn++) {  
        if(!camCtrlBnts[ctrlFn][0] && !camCtrlBnts[ctrlFn][1]) {  
            if(camAxisSpeedCmd[activeCam][ctrlFn] != ZERO) {  
                // Axis speed needs to be changed  
                camAxisSpeedCmd[activeCam][ctrlFn] = ZERO;  
                chgCamAxisSpd[activeCam][ctrlFn] = TRUE;  
            }  
        }  
        else {  
            if(camCtrlBnts[ctrlFn][0])  
                Move = MAX_SPEED_CCW; // Neg sense moves CCW  
            else  
                Move = MAX_SPEED_CW; // Pos sense moves CW  
  
            // Check button command. Issue new command to camera only  
            // when it's not already doing what operator has commanded.  
  
            // Button is on  
            if(camAxisSpeedCmd[activeCam][ctrlFn] != Move) {  
                // Axis speed needs to be changed  
                camAxisSpeedCmd[activeCam][ctrlFn] = Move;  
                chgCamAxisSpd[activeCam][ctrlFn] = TRUE;  
            }  
        }  
    }  
}
```



Target SW

```
}

// Check all camera motor commands for Stop conditions before sending
for(cam = 0; cam < NumCameras; cam++) {
    for(axis = 0; axis < NumCamAxis; axis++) {

        // Stop camera travel when axis limit has been reached

        if((camAngle[cam][axis] >= CamPosAngLimit[cam][axis] - 5) &&
           (camAngle[cam][axis] < CamPosAngLimit[cam][axis] + 90))
            // Axis is at positive limit, time to stop it moving in pos direction
            if(camAxisSpeedCmd[cam][axis] > ZERO) {
                camAxisSpeedCmd[cam][axis] = ZERO;
                chgCamAxisSpd[cam][axis] = TRUE;
            }
        if((camAngle[cam][axis] <= CamNegAngLimit[cam][axis] + 5) ||
           (camAngle[cam][axis] >= CamPosAngLimit[cam][axis] + 90))
            // Axis is at negative limit, time to stop it moving in neg direction
            if(camAxisSpeedCmd[cam][axis] < ZERO) {
                camAxisSpeedCmd[cam][axis] = ZERO;
                chgCamAxisSpd[cam][axis] = TRUE;
            }

        // Stop non-selected camera motors
        if(((cam != vidsrc1) && (cam != vidsrc2)) &&
           (camAxisSpeedCmd[cam][axis] != ZERO)) {
            camAxisSpeedCmd[cam][axis] = ZERO;
            chgCamAxisSpd[cam][axis] = TRUE;
        }

        // Send commands to active camera axis' as required
        if(chgCamAxisSpd[cam][axis]) {

            // Build & send command to motor
            *(afdxbase + 0x3000) = 0x0; // source address = RMS Computer
            *(afdxbase + 0x3001) = MotorVelocity; // Data type = motor velocity
            *(afdxbase + 0x3002) = camAxisSpeedCmd[cam][axis]; // Velocity
            *(afdxbase + 0x4000) = ExecuteCmd; // Command type
            *(afdxbase + 0x4001) = camMotorAFDXAddr[cam][axis]; // Issue command

            chgCamAxisSpd[cam][axis] = FALSE;
            freshCamAngle[cam][axis] = FALSE; // Time to get fresh angle data
        }
        // Retrieve updated axis angles from moving cameras
        if(!freshCamAngle[cam][axis]) {

            *(afdxbase + 0x3000) = 0x0; // source address = RMS Computer
            *(afdxbase + 0x3001) = JointAngle; // Data type = joint angle
            *(afdxbase + 0x4000) = QueryResponse; // Command type
            *(afdxbase + 0x4001) = camMotorAFDXAddr[cam][axis]; // Issue command

            camAngleRqstd[cam][axis] = TRUE;
        }
    }
}
```



Target SW

```
}

}

// Process camera lamp intensity change buttons
// Lamps auto increment/decrement with a period of LAMP_CHG_RATE * 100ms
// Percent incremented/decremented each period = LAMP_CHG_PCT

if(camCtrlBtns[Lights][Brt])
    // 'Lights Brt' button active
    if(++lampRateCnt >= LAMP_CHG_RATE) { // Auto-increment period
        lampRateCnt = 0;
        if(lampIntensity[activeCam] <= (100 - LAMP_CHG_PCT)) { // Intensity at max?
            // Increase lighting
            lampIntensity[activeCam] += LAMP_CHG_PCT;
            chgLampInten[activeCam] = TRUE;
        }
    }
    if(camCtrlBtns[Lights][Dim])
        // 'Lights Dim' button active
        if(++lampRateCnt >= LAMP_CHG_RATE) { // Auto-increment period
            lampRateCnt = 0;
            if(lampIntensity[activeCam] >= LAMP_CHG_PCT) { // Intensity at min?
                // Decrease lighting
                lampIntensity[activeCam] -= LAMP_CHG_PCT;
                chgLampInten[activeCam] = TRUE;
            }
        }
        // Send command to active camera lamp
        if(chgLampInten[activeCam]) {

            *(afdxbase + 0x3000) = 0x0; // source address = RMS Computer
            *(afdxbase + 0x3001) = Illumination; // Data type = Intensity
            *(afdxbase + 0x3002) = lampIntensity[activeCam]; // Intensity Pct
            *(afdxbase + 0x4000) = ExecuteCmd; // Command type
            *(afdxbase + 0x4001) = camLampAFDXAddr[activeCam]; // Load addx & send

            chgLampInten[activeCam] = FALSE;
        }
    }
```

ES Code

```
// Process camera control commands

if(camCtrlChg) { // Change camera movement only when directed
    camCtrlChg = FALSE;

    // Process Pan, Tilt, & Zoom control buttons
    // 0==Pan==Yaw, 1==Tilt==Pitch, 2==Zoom
    for(ctrlFn = Pan; ctrlFn <= Zoom; ctrlFn++) {
        if(!camCtrlBtns[ctrlFn][0] && !camCtrlBtns[ctrlFn][1]) {
            if(camAxisSpeedCmd[activeCam][ctrlFn] != ZERO) {
                /* Axis speed needs to be changed */
            }
        }
    }
}
```



ES Code

```
camAxisSpeedCmd[activeCam][ctrlFn] = ZERO;
chgCamAxisSpd[activeCam][ctrlFn] = TRUE;
}
}
else {
if(camCtrlBtns[ctrlFn][0])
Move = MAX_SPEED_CCW; // Neg sense moves CCW
else
Move = MAX_SPEED_CW; // Pos sense moves CW

// Check button command. Issue new command to camera only
// when it's not already doing what operator has commanded.

/* Button is on */
if(camAxisSpeedCmd[activeCam][ctrlFn] != Move) {
/* Axis speed needs to be changed */
camAxisSpeedCmd[activeCam][ctrlFn] = Move;
chgCamAxisSpd[activeCam][ctrlFn] = TRUE;
}
}
}

// Check all camera motor commands for Stop conditions before sending
for(cam = 0; cam < NumCameras; cam++) {
for(axis = 0; axis < NumCamAxis; axis++) {

// Stop camera travel when axis limit has been reached

if((camAngle[cam][axis] >= CamPosAngLimit[cam][axis] - 5) &&
(camAngle[cam][axis] < CamPosAngLimit[cam][axis] + 90))
// Axis is at positive limit, time to stop it moving in pos direction
if(camAxisSpeedCmd[cam][axis] > ZERO) {
camAxisSpeedCmd[cam][axis] = ZERO;
chgCamAxisSpd[cam][axis] = TRUE;
}
if((camAngle[cam][axis] <= CamNegAngLimit[cam][axis] + 5) ||
(camAngle[cam][axis] >= CamPosAngLimit[cam][axis] + 90))
// Axis is at negative limit, time to stop it moving in neg direction
if(camAxisSpeedCmd[cam][axis] < ZERO) {
camAxisSpeedCmd[cam][axis] = ZERO;
chgCamAxisSpd[cam][axis] = TRUE;
}
}

// Stop non-selected camera motors
if((cam != vidsrc1) && (cam != vidsrc2)) &&
(camAxisSpeedCmd[cam][axis] != ZERO)) {
camAxisSpeedCmd[cam][axis] = ZERO;
chgCamAxisSpd[cam][axis] = TRUE;
}

// Send commands to active camera axis' as required
if(chgCamAxisSpd[cam][axis]) {
```



ES Code

```
afdxsigout.msgType = AFDXSig::ExecuteCmd;
afdxsigout.address = camMotorAFDXAddr[cam][axis];
afdxsigout.data[1] = MotorVelocity; // Data type = Velocity
afdxsigout.data[2] = camAxisSpeedCmd[cam][axis]; // Velocity
SendSignal(Databus_A_O, &afdxsigout); // Issue command

chgCamAxisSpd[cam][axis] = FALSE;
freshCamAngle[cam][axis] = FALSE; // Time to get fresh angle data
}
// Retrieve updated axis angles from moving cameras
if(!freshCamAngle[cam][axis]/* && // Need to get fresh angle data?
!camAngleRqstd[cam][axis]*/) { // Await response from last query

afdxsigout.msgType = AFDXSig::QueryResponse; // Command type
afdxsigout.address = camMotorAFDXAddr[cam][axis]; // Set motor Addx
afdxsigout.data[1] = JointAngle; // Data type = joint angle
SendSignal(Databus_A_O, &afdxsigout); // Issue command

camAngleRqstd[cam][axis] = TRUE;
}
}

// Process camera lamp intensity change buttons
// Lamps auto increment/decrement with a period of LAMP_CHG_RATE * 100ms
// Percent incremented/decremented each period = LAMP_CHG_PCT

if(camCtrlBtns[Lights][Brt])
// 'Lights Brt' button active
if(++lampRateCnt >= LAMP_CHG_RATE) {
lampRateCnt = 0;
if(lampIntensity[activeCam] <= (100 - LAMP_CHG_PCT)) { // Intensity at max?
// Increase lighting
lampIntensity[activeCam] += LAMP_CHG_PCT;
chgLampInten[activeCam] = TRUE;
}
}
if(camCtrlBtns[Lights][Dim])
// 'Lights Dim' button active
if(++lampRateCnt >= LAMP_CHG_RATE) {
lampRateCnt = 0;
if(lampIntensity[activeCam] >= LAMP_CHG_PCT) { // Intensity at min?
// Decrease lighting
lampIntensity[activeCam] -= LAMP_CHG_PCT;
chgLampInten[activeCam] = TRUE;
}
}
// Send command to active camera lamp
if(chgLampInten[activeCam]) {

afdxsigout.msgType = AFDXSig::ExecuteCmd;
afdxsigout.address = camLampAFDXAddr[activeCam];
afdxsigout.data[1] = Illumination; // Data type = Intensity
```



ES Code

```
afdxsigout.data[2] = lampIntensity[activeCam]; // Intensity Pct
SendSignal(Databus_A_O, &afdxsigout); // Issue command

chgLampInten[activeCam] = FALSE;
}
```

3.6 Strain gauge, device fault, and video image management

This code segment receives data from the strain gauge modules, manages device fault reporting & clearing, and routes incoming camera video data as appropriate to the RMS Control Panel video monitors. Strain gauge response to forces has not been implemented but the mechanism by which the data modules are queried and data is reported is in place.

Since the target hardware design makes use of special ASICs for converting the digital bitmap images into RGB format, the target software loads the bitmap into the video ASIC which magically takes care of the conversion. Since the ES does not specify the implementation of how the bitmap image is converted to RGB, the data is ‘sent’ by simply passing pointers to the image buffer.

Using the DE part within the simulation slows the video displays down appreciably so code has been added to the software design to interlace the video image to compensate. This is not a requirement but merely an attempt to make the simulated displays refresh faster. In a real development project, this extra code might be used for early stages of testing, but would be removed as testing entered it’s final stages.

Target SW

```
// Process Strange Gage Information (when implemented)
// Send AFDX commands to query strain gauge data modules
for(ga = 0; ga < NumStrainGauges; ga++) {
    *(afdxbase + 0x3000) = 0x0; // source address = RMS Computer
    *(afdxbase + 0x3001) = Resistance; // Data type
    *(afdxbase + 0x4000) = QueryResponse; // Command
    *(afdxbase + 0x4001) = strainGaugeAFDXAddr[ga]; // Load addx & send
}

// Process Video Information
// Send AFDX commands to query all camera image sensors
for(cam = 0; cam < NumCameras; cam++) {
    *(afdxbase + 0x3000) = 0x0; // Source address
    *(afdxbase + 0x3001) = CamImage; // Data Type
    *(afdxbase + 0x4000) = QueryResponse; // Command
    *(afdxbase + 0x4001) = imageSensorAFDXAddr[cam]; // Load addx & send
}

// Process status data and update control panel fault lamps
cpFaultWord = 0;
for(joint = 0; joint < NumRMAJoints; joint++)
    if(rmaJointStatus[joint] != 0) { // Check for joint fault
        cpFaultWord |= faultMask[joint]; // Set joint fault bit
        cpFaultWord |= masterFaultMask; // Set master fault bit
    }
```



Target SW

```
for(cam = 0; cam < NumCameras; cam++) {
    if(imgSensorStatus[cam] != 0) // Check for image sensor fault
        cpFaultWord |= masterFaultMask; // Set master fault bit
    for(axis = 0; axis < NumCamAxis; axis++)
        if(camAxisStatus[cam][axis] != 0) // Check for cam axis fault
            cpFaultWord |= masterFaultMask; // Set master fault bit
}

for(ga = 0; ga < NumStrainGauges; ga++)
    if(strnGaModuleStatus[ga] != 0) // Check for strain gauge fault
        cpFaultWord |= masterFaultMask; // Set master fault bit

// If a fault's been reported or needs to be cleared, send fault word to control panel
if(cpFaultWord != lastFaultWd) { // Check to see if fault bit has changed

    iio->SetOutData(0, 0, (unsigned char)FaultData);
    iio->SetOutData(0, 1, 0); // Upper byte of message type always 0
    iio->SetOutData(0, 2, (unsigned char)cpFaultWord);
    iio->SetOutData(0, 3, (unsigned char)(cpFaultWord >> 8));

    fault_data = TRUE; // only one spi message allowed per frame

    lastFaultWd = cpFaultWord;
    cpFaultWord = 0; // Clear fault word
}

// Send the selected camera video to the correct control panel display

unsigned long tdataptr; // Temporary data pointer
unsigned char *video1base = (unsigned char *)0x03000000; // ASIC1 base addx
unsigned char *video2base = (unsigned char *)0x04000000; // ASIC2 base addx
int line, pixel;

// Get the Selected image for display 1
tdataptr = *(afdxbase + 0x2000 + imageSensorAFDXAddr[vidsrc1]);
unsigned short twobytes;

// Copy every 4th video line from the AFDX chip buffer to the video ASIC.
// Each pass through increments the line index until all lines of video have
// been sent. This video interlacing code helps to speed up the display
for(line = vidphase; line < 240; line += 4)
    for(pixel = 0; pixel < 320 * 4; pixel += 2) {
        i = 320 * 4 * line + pixel;
        twobytes = *((unsigned short *)((unsigned char *)afdxbase + tdataptr + i));
        *(video1base + i) = (unsigned char)twobytes;
        *(video1base + i + 1) = (unsigned char)(twobytes >> 8);
    }
*(video1base + 0x80000) = 0; // Triggers start of RGB conversion

// Get the Selected image for display 2
tdataptr = *(afdxbase + 0x2000 + imageSensorAFDXAddr[vidsrc2]);
```



Target SW

```
// Copy every 4th video line from the AFDX chip buffer to the video ASIC.  
// Each pass through increments the line index until all lines of video have  
// been sent. This video interlacing code helps to speed up the display  
for(line = vidphase; line < 240; line += 4)  
    for(pixel = 0; pixel < 320 * 4; pixel += 2) {  
        i = 320 * 4 * line + pixel;  
        twobytes = *((unsigned short *)((unsigned char *)afdxbase + tdataptr + i));  
        *(video2base + i) = (unsigned char)twobytes;  
        *(video2base + i + 1) = (unsigned char)(twobytes >> 8);  
    }  
*(video2base + 0x80000) = 0; // Triggers start of RGB conversion  
  
// Increment interlace line  
if(++vidphase >= 4)  
    vidphase = 0;
```

ES Code

```
// Process Strange Gage Information (when implemented)  
// Send AFDX commands to query strain gauge data modules  
for(ga = 0; ga < NumStrainGauges; ga++) {  
    afdxsigout.msgType = AFDXSig::QueryResponse; // Command type  
    afdxsigout.address = strainGaugeAFDXAddr[ga]; // Set AFDX Addx  
    afdxsigout.data[1] = Resistance; // Data type  
    SendSignal(Databus_A_O, &afdxsigout); // Issue command  
}  
  
// Process Video Information  
// Send AFDX commands to query all camera image sensors  
for(cam = 0; cam < NumCameras; cam++) {  
  
    afdxsigout.msgType = AFDXSig::QueryResponse; // Command type  
    afdxsigout.address = imageSensorAFDXAddr[cam]; // Dest addr  
    afdxsigout.data[1] = CamImage; // Data Type  
    SendSignal(Databus_A_O, &afdxsigout); // Send Command  
}  
  
// 3.3.2.3 Light or extinguish the control panel fault indicators  
// The RMS Computer shall process status data received from AFDX  
// devices and light the appropriate control panel fault indicator  
// when an AFDX device reports a fault. Each RMA joint has a dedicated  
// fault indicator in the RMA Joint Status window that shall be lit  
// when the associated RMA joint motor controller reports a fault.  
// When an RMA joint fault is reported, the Master Fault indicator  
// is also lit. All other faults reported by AFDX devices shall  
// light the Master Fault indicator only. All AFDX device status words  
// shall be continuously monitored so that when a fault is cleared,  
// the RMS computer will automatically extinguish the appropriate  
// fault indicator.  
  
// Process status data and update control panel fault indicators
```



ES Code

```
cpFaultWord = 0;
for(joint = 0; joint < NumRMAJoints; joint++)
    if(rmaJointStatus[joint] != 0) { // Check for joint fault
        cpFaultWord |= faultMask[joint]; // Set joint fault bit
        cpFaultWord |= masterFaultMask; // Set master fault bit
    }

for(cam = 0; cam < NumCameras; cam++) {
    if(imgSensorStatus[cam] != 0) // Check for image sensor fault
        cpFaultWord |= masterFaultMask; // Set master fault bit
    for(axis = 0; axis < NumCamAxis; axis++)
        if(camAxisStatus[cam][axis] != 0) // Check for cam axis fault
            cpFaultWord |= masterFaultMask; // Set master fault bit
}

for(ga = 0; ga < NumStrainGauges; ga++)
    if(strnGaModuleStatus[ga] != 0) // Check for strain gauge fault
        cpFaultWord |= masterFaultMask; // Set master fault bit

// If a fault needs to be reported or cleared, send fault word to control panel
if(cpFaultWord != lastFaultWd) { // Check to see if fault bit has changed

    bsig.tf = FALSE;
    SendSignal(_CS_DATA_1, &bsig);
    spisig.data = (short)FaultData;
    SendSignal(SerialChan_Data_1_Out, &spisig);
    spisig.data = (short)cpFaultWord;
    SendSignal(SerialChan_Data_1_Out, &spisig);
    bsig.tf = TRUE;
    SendSignal(_CS_DATA_1, &bsig);

    lastFaultWd = cpFaultWord;
    cpFaultWord = 0; // Clear fault word
}

// Point to video buffer for selected camera
rgbsig.bitmapptr = camBitMap[vidsrc1];

// Send video from selected camera to upper control panel display
SendSignal(RGB_1, &rgbsig);

// Point to video buffer for selected camera
rgbsig.bitmapptr = camBitMap[vidsrc2];

// Send video from selected camera to lower control panel display
SendSignal(RGB_2, &rgbsig);
```

3.7 Update RMA joint displays on Control Panel

This part of the code sends the current RMA joint angle, seek angle, and joint rate to the control panel for update of the RMA joint data display. The target software does not presently implement sending the fractional part of the displayed data as specified in the ES.



Beginning with the shoulder yaw joint, data is sent over the SPI bus in column order from Joint Angle through Joint Rate. Data for each successive RMA joint is updated until all data has been refreshed. Following update of the display, the velocity of each joint is recomputed.

Target SW

```
// Send joint angle, seek angle, and velocity data to control panel display

short bufPtr;
int n, past;
double angDist, currentAngle;

// Update joint angle history buffer
// This buffer is used to smooth out the velocity data through averaging
if(++angleHistCnt == 2) { // Angle updates every other time through processing loop
    angleHistCnt = 0;

    for(joint = 0; joint < NumRMAJoints; joint++) {
        angleHist[joint][current] = ((double)measAngle[joint]) + (((double)measAngleFract[joint]) / 4096); // Store
        current joint angle

        if(motorSpeedCmd[joint] == ZERO) { // Joint stopped, set buffer to present angle
            activeBuffSize[joint] = 1;

            // Set angle history buffer to current angle
            currentAngle = angleHist[joint][current];
            for(bufPtr = 0; bufPtr < ANGLE_HIST_SIZE; bufPtr++)
                angleHist[joint][bufPtr] = currentAngle;
        }
        else
            if(activeBuffSize[joint] < ANGLE_HIST_SIZE - 1)
                ++activeBuffSize[joint]; // Joint on the move, inc active buffer size
    }

    int tempAng;
    double angDiff;

    for(joint = 0; joint < NumRMAJoints; joint++) {

        // Compute velocity from angle history
        past = current;
        angDist = 0.0;

        for(n = 1; n <= activeBuffSize[joint]; n++) {

            past = current - n; // Adjust the past history buffer pointer
            if(past < 0) past += ANGLE_HIST_SIZE; // Circular buffer pointer mgmt

            angDiff = angleHist[joint][current] - angleHist[joint][past];
            tempAng = (int)(angDiff * 100.0); // *100 to keep precision after conv. to int

            // Simple zero crossing test
            // 288 is twice expected value @ 7.2 deg/sec
            if(tempAng > (288 * n))

```



Target SW

```
angDiff -= 360.0; // Adjust for CCW zero crossing
else if(tempAng < (-288 * n))
    angDiff +== 360.0; // Adjust for CW zero crossing

    angDist += (angDiff / (double)n); // Accumulate angle differences for averaging
}
// Angle history buffer updated every 42ms so measured velocity is multiplied by 1000 / 84
// Average the velocities and adjust for measurement interval
measVel[joint] = (angDist / ((double)(n - 1))) * (1000.0 / 84.0);

// Reasonableness test
if(fabs(measVel[joint]) > 15)
    measVel[joint] = lastVel[joint]; // Don't use unreasonable values
else lastVel[joint] = measVel[joint];
}
if(++current >= ANGLE_HIST_SIZE)
    current = 0;
}
if(!change_speed && !fault_data) {

int velInt, velFract;
double fint, fract;
double vel;

// send one set of joint data per frame
joint = frame_count % NumRMAJoints;

// Extract integer & fraction parts of dbl precision velocity to send over SPI bus
vel = measVel[joint];
fract = modf(vel, &fint); // modf defined above
velInt = (int)fint;
velFract = (int)(fract * 4096.0);

// Send RMA joint data to control panel over SPI bus
iio->SetOutData(0, 0, (unsigned char)PanelDisplayData);
iio->SetOutData(0, 1, 0); // Upper byte of PanelDisplayData always 0
iio->SetOutData(0, 2, (unsigned char)joint);
iio->SetOutData(0, 3, 0); // Upper byte of joint always 0
iio->SetOutData(0, 4, (unsigned char)measAngle[joint]);
iio->SetOutData(0, 5, (unsigned char)(measAngle[joint] >> 8));
iio->SetOutData(0, 6, (unsigned char)measAngleFract[joint]);
iio->SetOutData(0, 7, (unsigned char)(measAngleFract[joint] >> 8));
iio->SetOutData(0, 8, (unsigned char)seekAngle[joint]);
iio->SetOutData(0, 9, (unsigned char)(seekAngle[joint] >> 8));
iio->SetOutData(0, 10, (unsigned char)seekAngleFract[joint]);
iio->SetOutData(0, 11, (unsigned char)(seekAngleFract[joint] >> 8));
iio->SetOutData(0, 12, (unsigned char)velInt);
iio->SetOutData(0, 13, (unsigned char)(velInt >> 8));
iio->SetOutData(0, 14, (unsigned char)velFract);
iio->SetOutData(0, 15, (unsigned char)(velFract >> 8));
}
frame_count++;
```



ES Code

```
// 3.3.2.2 RMA joint status display control
// For each RMA joint the RMS Computer shall send the following data to the RMS
// control panel for display in the 'RMA Joint Status' window:
// Joint ID, Joint Angle, Seek Angle, Joint Velocity
// Display update rate shall be 200 ms.
// Joint angle & velocity data shall be floating point data sent as two 16-bit words
// containing first the integer value and second, the fraction value multiplied by 4096
// Joint velocity data shall be computed from a 1 second rolling average of joint angle data.

// Send joint angle, seek angle, and velocity data to control panel display

short bufPtr;
int n, past;
double angDist, currentAngle;

// Update joint angle history buffer
// This buffer is used to smooth out the velocity data through averaging
if(++angleHistCnt == 2) { // Angle updates every other time through processing loop
    angleHistCnt = 0;

    for(joint = 0; joint < NumRMAJoints; joint++) {
        angleHist[joint][current] = ((double)measAngle[joint]) + (((double)measAngleFract[joint]) / 4096); // Store
        current joint angle

        if(motorSpeedCmd[joint] == ZERO) { // Joint stopped, set buffer to present angle
            activeBuffSize[joint] = 1;

            // Set angle history buffer to current angle
            currentAngle = angleHist[joint][current];
            for(bufPtr = 0; bufPtr < ANGLE_HIST_SIZE; bufPtr++)
                angleHist[joint][bufPtr] = currentAngle;
        }
        else
            if(activeBuffSize[joint] < ANGLE_HIST_SIZE - 1)
                ++activeBuffSize[joint]; // Joint on the move, inc active buffer size
    }
    spisig.context = SPISig::ENDDATA;

    int velInt, velFract, tempAng;
    double fint, fract, angDiff;

    for(joint = 0; joint < NumRMAJoints; joint++) {

        // Compute velocity from angle history
        past = current;
        angDist = 0.0;

        for(n = 1; n <= activeBuffSize[joint]; n++) {

            past = current - n; // Adjust the past history buffer pointer
            if(past < 0) past += ANGLE_HIST_SIZE; // Circular buffer pointer mgmt
```



ES Code

```
angDiff = angleHist[joint][current] - angleHist[joint][past];
tempAng = (int)(angDiff * 100.0); // *100 to keep precision after conv. to int

// Simple zero crossing test
// 288 is twice expected value @ 7.2 deg/sec
if(tempAng > (288 * n))
    angDiff -= 360.0; // Adjust for CCW zero crossing
else if(tempAng < (-288 * n))
    angDiff += 360.0; // Adjust for CW zero crossing

angDist += (angDiff / (double)n); // Accumulate angle differences for averaging
}
// Angle history buffer updated every 200ms so measured velocity is multiplied by 5
// Average the velocities and adjust for measurement interval
measVel[joint] = (angDist / ((double)(n - 1))) * 5.0;

// Reasonableness test
if(fabs(measVel[joint]) > 15)
    measVel[joint] = lastVel[joint]; // Don't use unreasonable values
else lastVel[joint] = measVel[joint];

// Extract integer & fraction parts of dbl precision velocity to send over SPI bus
fract = modf(measVel[joint], &fint);
velInt = (int)fint;
velFract = (int)(fract * 4096.0);

// Send RMA joint data to control panel over SPI bus
bsig.tf = FALSE;
SendSignal(_CS_DATA_1, &bsig);
spisig.data = PanelDisplayData;
SendSignal(SerialChan_Data_1_Out, &spisig);
spisig.data = joint;
SendSignal(SerialChan_Data_1_Out, &spisig);
spisig.data = (short)measAngle[joint];
SendSignal(SerialChan_Data_1_Out, &spisig);
spisig.data = (short)measAngleFract[joint];
SendSignal(SerialChan_Data_1_Out, &spisig);
spisig.data = (short)seekAngle[joint];
SendSignal(SerialChan_Data_1_Out, &spisig);
spisig.data = (short)seekAngleFract[joint];
SendSignal(SerialChan_Data_1_Out, &spisig);
spisig.data = (short)velInt;
SendSignal(SerialChan_Data_1_Out, &spisig);
spisig.data = (short)velFract;
SendSignal(SerialChan_Data_1_Out, &spisig);
bsig.tf = TRUE;
SendSignal(_CS_DATA_1, &bsig);
}
if(++current >= ANGLE_HIST_SIZE)
    current = 0;
}
frame_count++;
```



3.8 Process incoming Control Panel data

The ExecuteCommand_() function is called when a new packet of data has been received from the control panel via the SPI receiver (see [paragraph 3.2](#)). This function processes the new button inputs and sets the appropriate variables, modes, and flags to act on the new information. The execution flow is straightforward with each of the major command codes being accommodated by a ‘case’ statement. The first four major commands primarily effect the joint-control state machine while the last two update changes to the desired camera view.

Target SW
<pre>void ControlProcess::ExecuteCommand() { unsigned short majorCommandCode; short joint; short ctrlFn, btn; short camCtrlWord; majorCommandCode = spiChan1Data[0]; // Retrieve new major command code switch(majorCommandCode) { case SelectMode: // Process control mode button inputs if(mode != spiChan1Data[1]) { // Potential mode change modeChange = TRUE; mode = spiChan1Data[1]; } break; case SelectMotor: // Process joint select button inputs selectedMotor = spiChan1Data[1]; break; case ManualCommand: // Process command button inputs cmdChange = TRUE; manualCmd = spiChan1Data[1]; break; case AngleSeek: // Process angle seek selection inputs joint = spiChan1Data[1]; seekAngle[joint] = spiChan1Data[2]; seekAngleFract[joint] = spiChan1Data[3]; newSeekAngle[joint] = TRUE; break; case Video1Select: // Process new camera source request for display 1 vidsrc1 = spiChan1Data[1]; break; case Video2Select: // Process new camera source request for display 2 vidsrc2 = spiChan1Data[1]; break; } }</pre>



Target SW

```
case Cam1Control: // Process display 1 camera control command inputs

camCtrlWord = spiChan1Data[1];

for(ctrlFn = 0; ctrlFn < NumCtrlFns; ctrlFn++) {
    for(btn = 0; btn < 2; btn++) {
        camCtrlBtns[ctrlFn][btn] = (1 & camCtrlWord); // Store LSB button state
        camCtrlWord >>= 1; // Shift next button state into LSB
    }

    camCtrlChg = TRUE;
    activeCam = vidsrc1;
    break;
}

case Cam2Control: // Process display 1 camera control command inputs

camCtrlWord = spiChan1Data[1];

for(ctrlFn = 0; ctrlFn < NumCtrlFns; ctrlFn++) {
    for(btn = 0; btn < 2; btn++) {
        camCtrlBtns[ctrlFn][btn] = (1 & camCtrlWord); // Store LSB button state
        camCtrlWord >>= 1; // Shift next button state into LSB
    }

    camCtrlChg = TRUE;
    activeCam = vidsrc2;
    break;
}
```

ES Code

```
// The ExecuteCommand function processes all SPI input bus data
// SPI input message formats are defined in the
// SW Design Requirements (SARP-I583-102) section 3.1

void RMSComputer::ExecuteCommand()
{
    unsigned short majorCommandCode;
    short joint;
    short ctrlFn, btn;
    short camCtrlWord;

    RGBSig rgbsig;
    AFDXSig afdxsigin;
    AFDXSig afdxsigout;

    majorCommandCode = spiChan1Data[0]; // Retrieve new data packet type code
    switch(majorCommandCode) {

        // 3.3.1 Mode Selection

        case SelectMode: // Process control mode button inputs
```



ES Code

```
if(mode != spiChan1Data[1]) { // Potential mode change
    modeChange = TRUE;
    mode = spiChan1Data[1];
}
break;

// 3.3.2 RMA joint motor selection inputs

case SelectMotor: // Process joint select button inputs
    selectedMotor = spiChan1Data[1];
    break;

// 3.3.3 Manual Control Inputs

case ManualCommand: // Process command button inputs
    cmdChange = TRUE;
    manualCmd = spiChan1Data[1];
    break;

// 3.3.4 Angle Seek Inputs

case AngleSeek: // Process angle seek selection inputs
    joint = spiChan1Data[1];
    seekAngle[joint] = spiChan1Data[2];
    seekAngleFract[joint] = spiChan1Data[3];
    newSeekAngle[joint] = TRUE;

    break;

// 3.3.5 Video display monitor camera source selection

case Video1Select: // Process new camera source request for display 1
    vidsrc1 = spiChan1Data[1];

    // Point to video buffer for selected camera
    rgbsig.bitmapptr = camBitMap[vidsrc1];

    // Send video from selected camera to upper control panel display
    SendSignal(RGB_1, &rgbsig);
    break;

case Video2Select: // Process new camera source request for display 2
    vidsrc2 = spiChan1Data[1];

    // Point to video buffer for selected camera
    rgbsig.bitmapptr = camBitMap[vidsrc2];

    // Send video from selected camera to lower control panel display
    SendSignal(RGB_2, &rgbsig);
    break;

// 3.3.6 Video display monitor active camera control input
```



ES Code

```
case Cam1Control: // Process display 1 camera control command inputs

    camCtrlWord = spiChan1Data[1];

    for(ctrlFn = 0; ctrlFn < NumCtrlFns; ctrlFn++)
        for(btn = 0; btn < 2; btn++) {
            camCtrlBtns[ctrlFn][btn] = (1 & camCtrlWord); // Store LSB button state
            camCtrlWord >>= 1; // Shift next button state into LSB
        }

    camCtrlChg = TRUE;
    activeCam = vidsrc1;
    break;

case Cam2Control: // Process display 1 camera control command inputs

    camCtrlWord = spiChan1Data[1];

    for(ctrlFn = 0; ctrlFn < NumCtrlFns; ctrlFn++)
        for(btn = 0; btn < 2; btn++) {
            camCtrlBtns[ctrlFn][btn] = (1 & camCtrlWord); // Store LSB button state
            camCtrlWord >>= 1; // Shift next button state into LSB
        }

    camCtrlChg = TRUE;
    activeCam = vidsrc2;
    break;
}
```

3.9 Header Files

The ‘C’ language header file excerpts in this section have been included for reference in understanding data structures, constants, and program flow of the target SW and ES code listed in the preceding sections. The target SW uses the ControlProcess.h file, the ES code uses the RMSComputer.h file, and the target SW and the ES code both use the ControlPanelICD.h file.

ControlPanel_ICD.h Excerpts

```
// Control Panel Command ICD
// Rev-

// Commands from Control Panel to Computer

#ifndef __CONTROLPANEL_ICD
#define __CONTROLPANEL_ICD

enum MajorCommands {
    SelectMode,
    SelectMotor,
```



ControlPanel_ICD.h Excerpts

```
ManualCommand,  
AngleSeek,  
Video1Select,  
Video2Select,  
Cam1Control, // Camera1 control command  
Cam2Control, // Camera2 control command  
};  
enum Cameras {  
    UpperArmCam,  
    LowerArmCam,  
    WristCam,  
    FwdBayCam,  
    AftBayCam,  
    NumCameras,  
};  
enum ControlModes {  
    AngleSeekCM,  
    ManualCM,  
};  
enum ManualCommands {  
    INC,  
    DEC,  
    STOP,  
    CmdBtnsOff,  
};  
  
// Camera Control buttons  
enum CamControlFuncts {  
    Pan,  
    Tilt,  
    Zoom,  
    Lights,  
  
    NumCtrlFns,  
};  
enum { /* Pan buttons */  
    Lt,  
    Rt,  
};  
enum { /* Tilt buttons */  
    Up,  
    Dn,  
};  
enum { /* Zoom buttons */  
    In,  
    Out,  
};  
enum { /* Light buttons */  
    Dim,  
    Brt,  
};
```



ControlPanel_ICD.h Excerpts

```
// Data from Computer to Control Panel
```

```
enum DataGroups {  
    PanelDisplayData,  
    ManualCmdData,  
    FaultData,  
};
```

```
enum SwitchBacklight {  
    BacklightOff,  
    BacklightOn,  
};
```

```
enum FaultDataBits {  
    MasterFault = 1,  
    shldYawFault = 2,  
    shldPitchFault = 4,  
    elbowPitchFault = 8,  
    wristPitchFault = 0x10,  
    wristYawFault = 0x20,  
    wristRollFault = 0x40,  
};
```

```
enum DisplayDataIDs {  
    DispJointAng,  
    DispSeekAng,  
    DispJointRate,
```

```
    NumColumns,  
};
```

```
enum RMAJoints {  
    ShoulderYaw,  
    ShoulderPitch,  
    ElbowPitch,  
    WristPitch,  
    WristYaw,  
    WristRoll,
```

```
    NumRMAJoints,  
};
```

```
enum CameraAxis {  
    Yaw,  
    Pitch,  
    Zoom_,
```

```
    NumCamAxis,  
};
```

ControlProcess.h Excerpts

```
enum constant {  
    ZERO = 0,
```



ControlProcess.h Excerpts

```
MOTOR_SPEED = 600, // Speed for moving all motors (in RPM)
    // Joint RPM = Motor RPM / 500 = 7.2 deg/sec
MAX_SPEED_CW = MOTOR_SPEED,
MAX_SPEED_CCW = -MOTOR_SPEED,

LAMP_CHG_RATE = 3, // Intensity change interval = n * 100ms
LAMP_CHG_PCT = 5, // Lamp intensity change percent inc (or dec)

ANGLE_HIST_SIZE = 6, // Angle history buff used for averaging joint vel

MAX_SPI_WORDS = 16, // Maximum SPI bus data packet size

AFDX_PKT_SIZE = 16, // Maximum AFDX bus data packet size

NUM_CHANNELS = 4, // Num of analog channels per data module
};

enum RMAJointControlStates {
    GetJointAngle,
    Seeking,
    ProcessCmd,
    ChangeSpeed,
};

enum SeekDirection {
    CW,
    CCW,
};

enum StrainGauges {
    UpperArm,
    LowerArm,
    NumStrainGauges,
};

enum AFDX_MessageTypes {
    None,
    QueryStatusAll,
    QueryResponse,
    ExecuteCmd,
};

enum AFDX_DataTypes {

    // All devices
    Status = 1,
    SelfTest,
    CmdError,

    // Joint & camera motor controller data
    MotorVelocity,
    MotorAngle,
```



ControlProcess.h Excerpts

```
JointAngle,  
  
// Strain gauge data module  
Resistance,  
  
// Camera lamps  
Illumination,  
  
// Image Sensor  
CamImage,  
};  
  
// Definition of command error codes  
enum CmdErrorCodes {  
    InvalidChNum = 1, // Data module  
};  
  
// Data module status bits  
enum DataModStatusCodes {  
    ResistanceFault = 1,  
    DataModuleFault = 2,  
};  
  
// Motor controller status bits  
enum MotorStatusCodes {  
    MotorFault = 1,  
    ControllerFault = 2,  
    NobodysFault = 4,  
    YourFault = 8,  
    MyFault = 0x10,  
    SanAndreasFault = 0x20,  
    TowerFawlyt = 0x40,  
};  
  
// Image sensor status bits  
enum ImageSensorStatusIDs {  
    SensorFault = 1,  
    Tarnished = 2,  
    Faded = 4,  
    LargerThanLife = 8,  
};  
  
BOOL resetFault;  
unsigned int cpFaultWord;  
unsigned int lastFaultWd;  
unsigned int masterFaultMask;  
unsigned int faultMask[NumRMAJoints];  
  
// AFDX device address storage arrays  
unsigned long rmaMotorAFDXAddr[NumRMAJoints];
```



ControlProcess.h Excerpts

```
unsigned long imageSensorAFDXAddr[NumCameras];
unsigned long camMotorAFDXAddr[NumCameras][NumCamAxis];
unsigned long camLampAFDXAddr[NumCameras];
unsigned long strainGaugeAFDXAddr[NumStrainGauges];

unsigned long *afdxbase;

// Status word storage
unsigned int rmaJointStatus[NumRMAJoints];
unsigned int camAxisStatus[NumCameras][NumCamAxis];
unsigned int imgSensorStatus[NumCameras];
unsigned int strnGaModuleStatus[NumStrainGauges];

// Strain Gauge variables
int strnGaData[NumStrainGauges][NUM_CHANNELS];
int errorMsg;
int analogChanNum; // Each data module has 4 analog channels

// RMA joint variables
BOOL jointAngleRqstd[NumRMAJoints];
BOOL freshAngleData[NumRMAJoints];
BOOL newSeekAngle[NumRMAJoints];
short motorSpeedCmd[NumRMAJoints];
int measAngle[NumRMAJoints];
int measAngleFract[NumRMAJoints];
int seekAngle[NumRMAJoints];
int seekAngleFract[NumRMAJoints];
int fill1[100];
short activeBuffSize[NumRMAJoints];
double angleHist[NumRMAJoints][ANGLE_HIST_SIZE];
double measVel[NumRMAJoints];
double lastVel[NumRMAJoints];
int fill2[100];
unsigned short jointControlState[NumRMAJoints];

// Camera motor variables
BOOL freshCamAngle[NumCameras][NumCamAxis];
BOOL camAngleRqstd[NumCameras][NumCamAxis];
BOOL chgCamAxisSpd[NumCameras][NumCamAxis];
short camAxisSpeedCmd[NumCameras][NumCamAxis];
int camAngle[NumCameras][NumCamAxis];
unsigned short CamPosAngLimit[NumCameras][NumCamAxis];
unsigned short CamNegAngLimit[NumCameras][NumCamAxis];
short camCtrlBtns[NumCtrlFns][2];

// Camera lamp variables
BOOL chgLampInten[NumCameras];
short lampIntensity[NumCameras];

// Camera video data pointers & variables
unsigned char *camBitMap[NumCameras];
BOOL camBitMapFresh[NumCameras];
```



ControlProcess.h Excerpts

```
short spiChan1Data[MAX_SPI_WORDS];

BOOL rtos_init;
int frame_count;

short wordcount;
int angleUpdateCnt;
int velocityUpdateCnt;
int current;
short angleHistCnt;
unsigned short lampRateCnt;

BOOL modeChange;
BOOL cmdChange;
BOOL camCtrlChg;

unsigned short mode;
unsigned short selectedMotor;
unsigned short manualCmd;

unsigned short displayJoint;
unsigned short displayDataID;
unsigned short vidsrc1;
unsigned short vidsrc2;
unsigned short activeCam;

unsigned short vidphase;
```

RMSComputer.h Excerpts

```
// put member declarations here

enum constant {
    ZERO = 0,

    MOTOR_SPEED = 600, // Speed for moving all motors (in RPM)
        // Joint RPM = Motor RPM / 500 = 7.2 deg/sec
    MAX_SPEED_CW = MOTOR_SPEED,
    MAX_SPEED_CCW = -MOTOR_SPEED,

    LAMP_CHG_RATE = 3, // Intensity change interval = n * 100ms
    LAMP_CHG_PCT = 5, // Lamp intensity change percent inc (or dec)

    ANGLE_HIST_SIZE = 6, // Angle history buff used for averaging joint vel

    MAX_SPI_WORDS = 16, // Maximum SPI bus data packet size

    AFDX_PKT_SIZE = 16, // Maximum AFDX bus data packet size

    NUM_CHANNELS = 4, // Num of analog channels per data module
};
```



RMSComputer.h Excerpts

```
enum RMAJointControlStates {
    GetJointAngle,
    Seeking,
    ProcessCmd,
    ChangeSpeed,
};

enum SeekDirection {
    CW,
    CCW,
};

enum StrainGauges {
    UpperArm,
    LowerArm,
    NumStrainGauges,
};

enum AFDX_MessageTypes {
    None,
    QueryStatusAll,
    QueryResponse,
    ExecuteCmd,
};

enum AFDX_DataTypes {

    // All devices
    Status = 1,
    SelfTest,
    CmdError,

    // Joint & camera motor controller data
    MotorVelocity,
    MotorAngle,
    JointAngle,

    // Strain gauge data module
    Resistance,

    // Camera lamps
    Illumination,

    // Image Sensor
    CamImage,
};

// Definition of command error codes
enum CmdErrorCodes {
    InvalidChNum = 1, // Data module
};
```



RMSComputer.h Excerpts

```
// Data module status bits
enum DataModStatusIDs {
    ResistanceFault = 1,
    DataModuleFault = 2,
};

// Motor controller status bits
enum MotorStatusIDs {
    MotorFault = 1,
    ControllerFault = 2,
    NobodysFault = 4,
    YourFault = 8,
    MyFault =0x10,
    SanAndreasFault = 0x20,
    TowerFawlty = 0x40,
};

// Image sensor status bits
enum ImageSensorStatusIDs {
    SensorFault = 1,
    Tarnished = 2,
    Faded = 4,
    LargerThanLife = 8,
};

BOOL resetFault;
unsigned int cpFaultWord;
unsigned int lastFaultWd;
unsigned int masterFaultMask;
unsigned int faultMask[NumRMAJoints];

// AFDX device address storage arrays
unsigned long rmaMotorAFDXAddr[NumRMAJoints];
unsigned long imageSensorAFDXAddr[NumCameras];
unsigned long camMotorAFDXAddr[NumCameras][NumCamAxis];
unsigned long camLampAFDXAddr[NumCameras];
unsigned long strainGaugeAFDXAddr[NumStrainGauges];

// Status word storage
unsigned int rmaJointStatus[NumRMAJoints];
unsigned int camAxisStatus[NumCameras][NumCamAxis];
unsigned int imgSensorStatus[NumCameras];
unsigned int strnGaModuleStatus[NumStrainGauges];

// Strain Gauge variables
int strnGaData[NumStrainGauges][NUM_CHANNELS];
int errorMsg;
int analogChanNum; // Each data module has 4 analog channels

// RMA joint variables
```



RMSComputer.h Excerpts

```
BOOL jointAngleRqstd[NumRMAJoints];
BOOL freshAngleData[NumRMAJoints];
BOOL newSeekAngle[NumRMAJoints];
short motorSpeedCmd[NumRMAJoints];
int measAngle[NumRMAJoints];
int measAngleFract[NumRMAJoints];
int seekAngle[NumRMAJoints];
int seekAngleFract[NumRMAJoints];
short activeBuffSize[NumRMAJoints];
double angleHist[NumRMAJoints][ANGLE_HIST_SIZE];
double measVel[NumRMAJoints];
double lastVel[NumRMAJoints];
unsigned short jointControlState[NumRMAJoints];

// Camera motor variables
BOOL freshCamAngle[NumCameras][NumCamAxis];
BOOL camAngleRqstd[NumCameras][NumCamAxis];
BOOL chgCamAxisSpd[NumCameras][NumCamAxis];
short camAxisSpeedCmd[NumCameras][NumCamAxis];
int camAngle[NumCameras][NumCamAxis];
unsigned short CamPosAngLimit[NumCameras][NumCamAxis];
unsigned short CamNegAngLimit[NumCameras][NumCamAxis];
short camCtrlBtns[NumCtrlFns][2];

// Camera lamp variables
BOOL chgLampInten[NumCameras];
short lampIntensity[NumCameras];

// Camera video data pointers & variables
unsigned char *camBitMap[NumCameras];
BOOL camBitMapFresh[NumCameras];

short spiChan1Data[MAX_SPI_WORDS];

BOOL rtos_init;
int frame_count;

BOOL _cs_1;
short wordcount;
int angleUpdateCnt;
int velocityUpdateCnt;
int current;
short angleHistCnt;
unsigned short lampRateCnt;

BOOL modeChange;
BOOL cmdChange;
BOOL camCtrlChg;

unsigned short mode;
unsigned short selectedMotor;
```



RMSComputer.h Excerpts

```
unsigned short manualCmd;  
  
unsigned short displayJoint;  
unsigned short displayDataID;  
unsigned short vidsrc1;  
unsigned short vidsrc2;  
unsigned short activeCam;
```